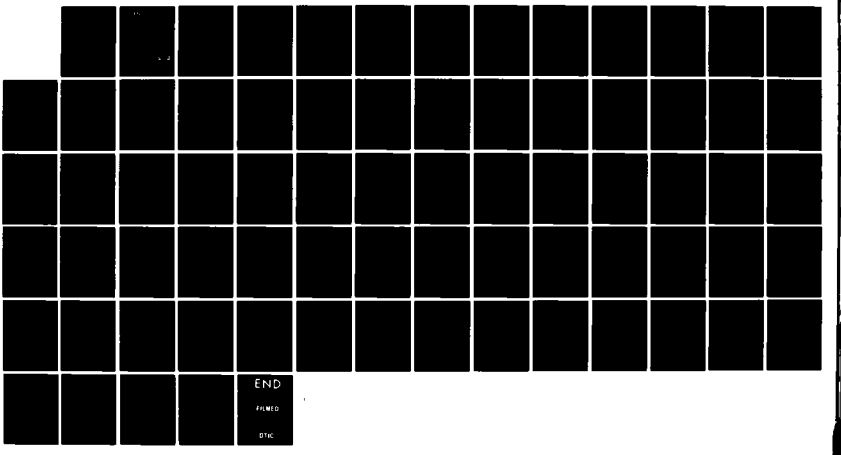
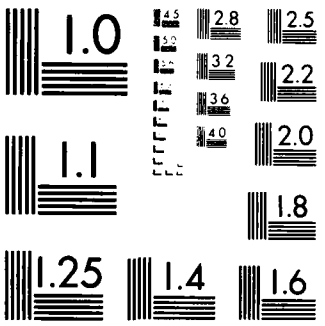
AD-A149 323 DESIGN OF A PICTORIAL PROGRAM REFERENCE LANGUAGE(U) 1/1
ADVANCED INFORMATION AND DECISION SYSTEMS MOUNTAIN VIEW
CA E A DOMESHEK ET AL. AUG 84 AI/DS-TM-1014-4
UNCLASSIFIED AFOSR-TR-84-1159 F49620-81-C-0067 F/G 9/2 NL





AI&DS

13

Final Report
TM-1014-4

DESIGN OF A PICTORIAL PROGRAM REFERENCE LANGUAGE

Eric A. Domeshek
Jeffrey S. Dean
Susan G. Rosenbaum
Brian P. McCune

Advanced Information & Decision Systems
201 San Antonio Circle, Suite 286
Mountain View, CA 94040-1270

August 1984

Final Technical Report for June 1, 1981 - 30 May, 1984

Approved for public release; distribution unlimited

Prepared for:

United States Air Force
Air Force Office of Scientific Research
Building 410
Bolling Air Force Base, D.C. 20332

ADVANCED INFORMATION & DECISION SYSTEMS

Mountain View, CA 94040

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited.
2b DECLASSIFICATION DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S): TM-1014-4	5. MONITORING ORGANIZATION REPORT NUMBER(S): AFOSR-TR- 84 1159
6a NAME OF PERFORMING ORGANIZATION: Advanced Information and Decision Systems; 6b OFFICE SYMBOL (If applicable)	7a NAME OF MONITORING ORGANIZATION: Air Force Office of Scientific Research
6c ADDRESS (City, State and ZIP Code): 201 San Antonio Circle, Suite 286, Mountain View CA 94040-1270	7b ADDRESS (City, State and ZIP Code): Directorate of Mathematical and Information Sciences, Bolling AFB DC 20332
8a NAME OF FUNDING/SPONSORING ORGANIZATION: AFOSR; 8b OFFICE SYMBOL (If applicable): NM	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: F49620-81-C-0067
8c ADDRESS (City, State and ZIP Code): Bolling AFB DC 20332	10 SOURCE OF FUNDING NOS: PROGRAM ELEMENT NO. 61102F; PROJECT NO. 2304; TASK NO A7; WORK UNIT NO

11 TITLE (Include Security Classification): DESIGN OF A PICTORIAL PROGRAM REFERENCE LANGUAGE

12 PERSONAL AUTHOR(S): Eric A. Domeshek, Jeffrey S. Dean, Susan G. Rosenbaum, and Brian P. McCune.

13a TYPE OF REPORT	13b. TIME COVERED	14 DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
Final	FROM 1/6/83 TO 31/5/84	31 AUG 84	70

16. SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB GR	Program Reference Language (PRL); Extended Program Model (EPM); Intelligent Program Editor (IPE); program documentation; artificial intelligence (AI); CONTINUED

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

This report covers the work done during the third year of the Program Reference Language (PRL) project. During this year we focused on the problem of developing an adequate means to express the types of queries we had earlier identified as within the province of the PRL. Thus, we studied both the structure of the actual query language and the design of the user interface. The query language on which we concentrated was a pictorial interface designated the PRL Pictorial Language (PRL/PL). The essential idea is that the users build templates to sketch out what an item that satisfied the query would look like. For programs, this means specifying an arrangement of standard program fragments that characterize the desired part of the program. This document will discuss the design and use of this pictorial language.

20 DISTRIBUTION AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS ☐	UNCLASSIFIED

22a NAME OF RESPONSIBLE INDIVIDUAL	22b TELEPHONE NUMBER (Include Area Code)	22c OFFICE SYMBOL
Dr. Robert N. Buchal	(202) 767- 4939	NM

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE

ITEM #18, SUBJECT TERMS, CONTINUED: knowledge base; multiple representations; protocol analysis; user modeling; retrieval language; debugging program cliches; program annotations.

TABLE OF CONTENTS

Chief, Tec[illegible]formation Division

LIST OF FIGURES

1. INTRODUCTION

This report documents the third year of work on the Program Reference Language project (PRL), which is a basic research effort aimed at the creation of a mechanism for flexibly identifying the interesting portions of programs. During this year we focused on the issue of developing a means of expressing the types of queries identified earlier as within the province of the PRL. We studied both the structure of the actual query language and the design of the user interface.

The PRL is designed to allow the user to describe a piece of a program so that an automated search mechanism can retrieve any matching program fragments. An extended PRL might also allow the user to specify transformations to be performed on a selected set of program fragments, but until we have time to develop and classify a useful set of such transformations, we consider only the problem of specifying and performing searches. The work which preceded this study is discussed in length in the annual reports for the first and second years of research and will be recapped only briefly below. (See "Searching a Knowledge Base of Programs and Documentation," [Shapiro-83] and "An Informal Study of Program Comprehension," [Domeshek-84] for more details.) This document focuses on the design of the PRL Picture Language, the pictorial interface to the underlying database.

1.1 OVERVIEW

Earlier PRL research led to the definition of the Extended Program Model (EPM) [Shapiro-83], a database which holds multiple representations of computer programs. The particular forms in which programs were to be stored, the types of analyses required to generate these representations, and the information made available by these analyses were considered. Viewing the EPM as a database leads naturally to a view of the PRL as a database query language. Our goals for the PRL, combined with our design for the EPM, guided us in the design of the query language.

Study of existing database query languages indicated that the task of designing a formal query language would not be trivial. Although the definition of a language capable of expressing the required searches might be straightforward, such a language would not be easily usable by programmers. However, such a formal basis is essential for the PRL, if only for internal use. Thus, while some time was spent studying the issues of a formal query language, the majority of the effort was spent investigating the possibilities of two options for more "friendly" query languages that could be translated to the formal language.

The first option considered was a limited natural language interface that was designated the PRL Natural Language (PRL/NL). There are already several natural language interfaces designed for database applications, and early PRL examples had always phrased sample queries in this manner. Our own experience with English renderings of PRL queries caused us to believe that they tended to obscure the regularity of the actual relationships being expressed. A survey of the literature revealed many problems complicating the design of a compact and

consistent subset of English intended for use as a query language.

The second option, and the one on which we concentrated, was a pictorial interface, designated the PRL Picture Language (PRL/PL). This approach was originally inspired by the Query-By-Example (QBE) database query language [Zloof-1977]; however, the Picture Language is considerably different from QBE. In the PRL, obvious specific knowledge about the structure of the database has been incorporated to increase its power and decrease its complexity. The essential idea behind QBE and PRL/PL is that a user partially sketches out the picture of the requested item. For the PRL, this means specifying an arrangement of program fragments that characterizes the desired part of the program.

1.2 RESEARCH OBJECTIVES

Some of the key research issues driving the PRL effort are:

1. What are the most useful ways of referring to parts of a program? Said in a different way, what vocabulary do programmers currently use to describe portions of their programs?
2. What information must be included in a knowledge base about programs and documentation in order for it to support program search?
3. What information must be included in such a knowledge base for it to support a variety of intelligent tools for accessing and manipulating code?
4. How should information of this kind be represented?
5. How should application specific knowledge be included?
6. How can user-supplied assertions and other documentation be acquired and integrated into a knowledge base for use in program referencing and other tasks?

7. How can search requests be expressed in a uniform reference language?

8. What form of a search mechanism is required to implement these reference requests?

9. How can these searches be performed efficiently? In what ways can search be limited or deferred in order to maintain good response time?

1.3 GUIDE TO READING

The following sections provide details about the PRL. Section 2 introduces the PRL Picture Language (PRL/PL) and gives examples of its usage. Section 3 discusses the PRL formal language (PRL/FL). Section 4 provides a description of the editor interface for the PRL/PL; section 5 discusses some of the problems still remaining with the PRL/PL and PRL/FL and our future research plans. This report concludes with discussion of key research personnel and their activities.

2. PRL PICTURE LANGUAGE: PRL/PL

2.1 INTRODUCTION TO PRL PICTURE LANGUAGE

The PRL Picture Language provides a user friendly interface, making it easy for programmers to specify searches through pr[illegible] freely combining information from the multiple representations of the program maintained by the Extended Program Model, or EPM. There is a formal mapping from these pictures to the PRL Formal Language. An underlying operational semantics for the Formal Language thus provides a formal means for interpreting pictures expressed in the Picture Language.

The PRL/PL is similar to QBE in that a query is specified by describing a typical item that would satisfy the query; however, as the name implies, the interface is picture oriented. QBE is designed for relational databases; its templates are partially filled-in tables representing the various relations. Due to the internal tree/network format underlying the EPM database, the PRL is not well suited to the relational model. The templates for PRL/PL queries reflect this network structure in that a major form of composition is nesting of boxes inside other boxes.

The PRL/PL is intended to be intuitive and easy to use for typical simple requests; however, it shares the problem found in QBE in that complex requests

begin to require complex templates. Since the spatial relations of pieces of the PRL/PL query tend to be meaningful, though, this problem should not be as severe. With the PRL/PL, the query is composed of boxes that form a picture of the overall shape of the query, while in QBE, the user must work with many separate tables representing the database relations.

The goal in designing a pictorial query language is to ensure that any query specifiable in the formal language has at least one pictorial representation. The PRL/PL must:

- Provide a mapping to the formal query language
- Be simple and intuitive
- Retain the integrity of pictorial query fragments across different contexts, which implies a maintaining of composability across different environments

2.2 DESCRIPTION OF PRL/PL

Picture queries are composed of boxes representing fragments of programs. The vocabulary of the PRL/PL consists of all of the program fragments understood by the EPM. Each fragment is represented by a box. The basic operator of composition is **containment**; if a picture shows a box inside another box, this specifies a match against the database in which the fragments of code have the same relationship.

The picture representing the query "Find the FUNCTIONS that contain LOOPS" is shown in figure 1. Note that there is a box of type *function* with a box of type *loop* inside of it. Fragments of code which are function definitions that include loops will match this search template.

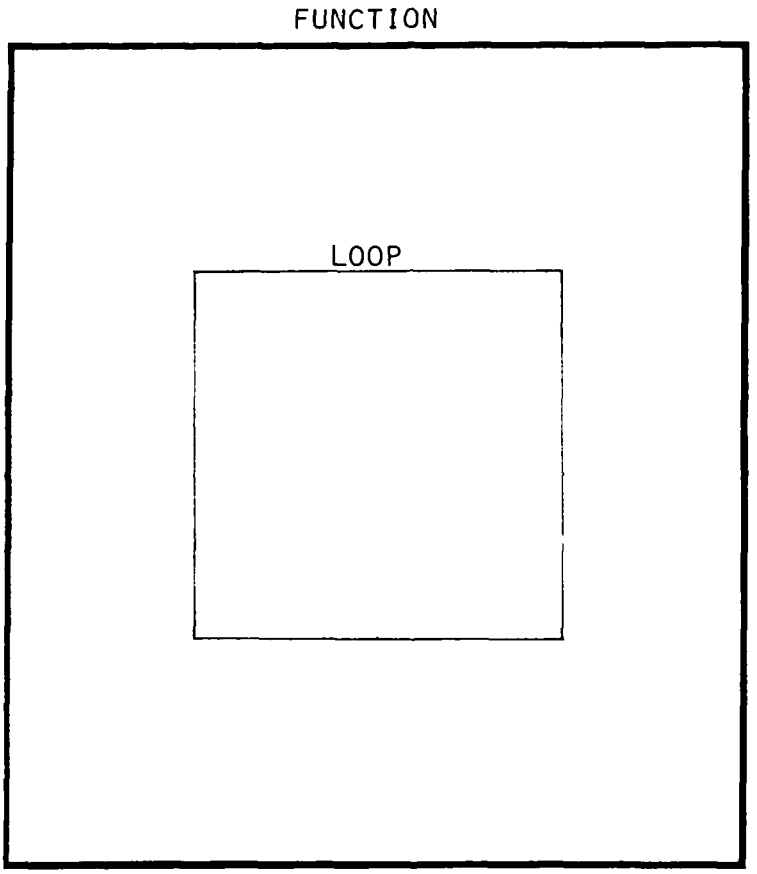


Figure 1. FIND THE FUNCTIONS THAT CONTAIN LOOPS

A convention used in PRL/PL pictures is that the object to be returned as a result of the query is shown in a highlighted box. Consider the alternative query "Find all LOOPS contained in FUNCTIONS" shown in figure 2. Note that the box representing the *loop* is now drawn with heavier lines. This query

will return a set of loops as its result.

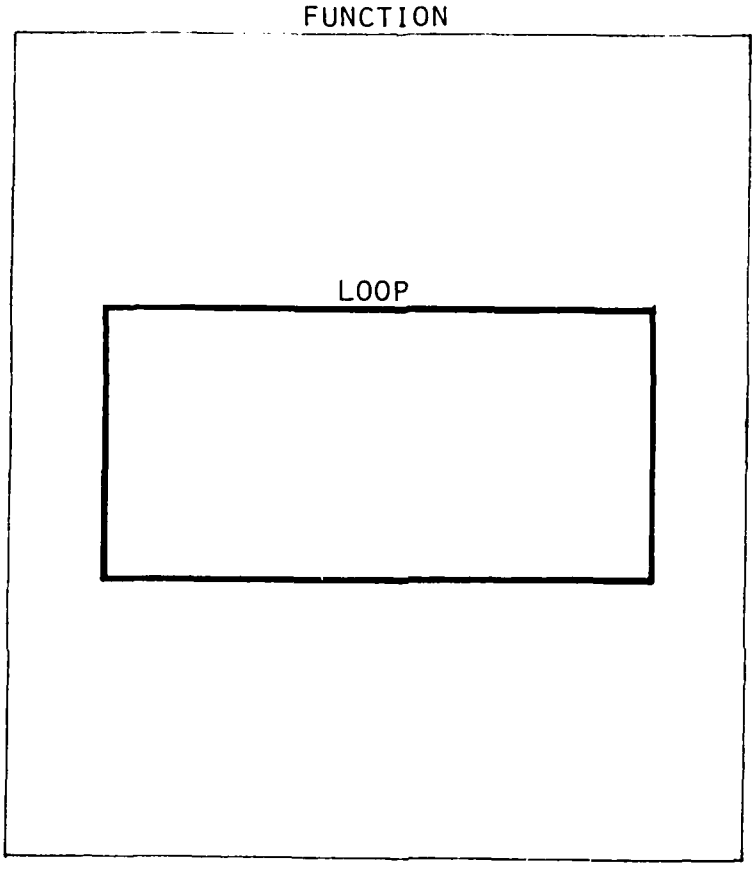


Figure 2: FIND THE LOOPS CONTAINED IN FUNCTIONS

Queries can be much more complex than these first examples. We will survey the variations and additional features of the PRL/PL, introducing each new feature with an illustrative example.

It is possible to specify that an object contain more than one object. The basic form of such a compound containment is illustrated in figure 3, the picture representing the query "Find all FUNCTIONS containing LOOPS and IF-

STATEMENTS." The box representing the function now contains two other boxes, one representing a *loop*, the other representing an *if-statement*. The search is again intended to return a set of functions, but now only those that contain both a *loop* and an *if-statement* are valid matches. Note that there is an implicit conjunction of the inner boxes.

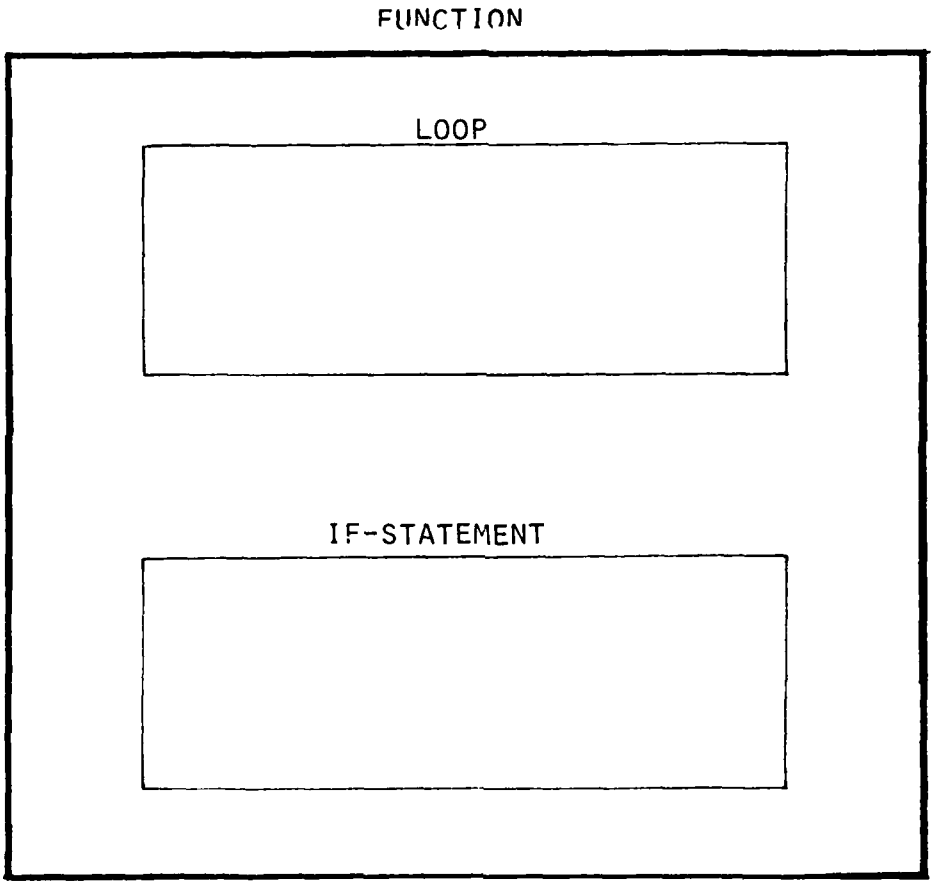


Figure 3: FIND ALL FUNCTIONS CONTAINING LOOPS AND IF-STATEMENTS

If we wanted to perform the search "Find all FUNCTIONS containing a *LOOP followed by an IF-STATEMENT*," the graphical query would look like

figure 4. Note the arrow drawn from the box representing the *loop* to the box representing the *if-statement*; this is the pictorial representation of precedence, or textual ordering. It may be possible to extend this idea to represent flow ordering, as this also might be a useful search constraint and the information can be generated from the EPM. Precedence relations are restricted to apply to objects taking part in a conjunction. Note that without the arrow, figure 4 becomes identical to figure 3.

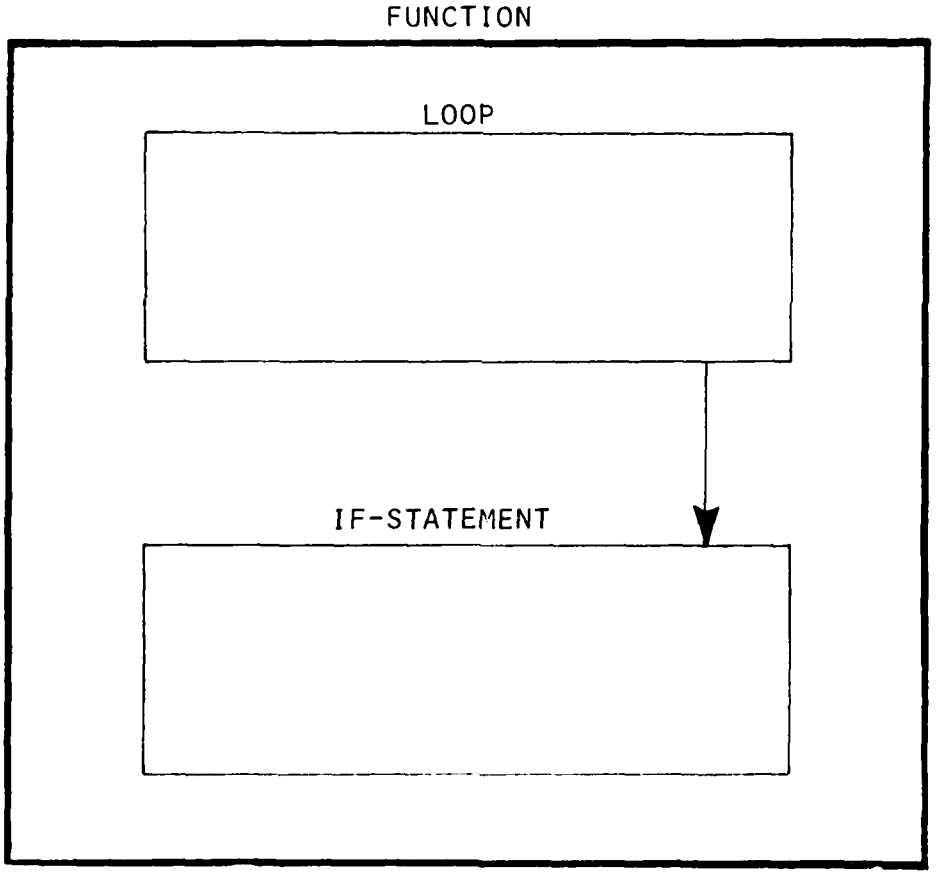


Figure 4: FIND ALL FUNCTIONS CONTAINING A LOOP FOLLOWED BY AN IF-STATEMENT

The nesting of boxes within boxes is not limited to a single level. For example, the query, "Find all FUNCTIONS which contain LOOPS that contain IF-STATEMENTS" is illustrated in figure 5. Again, multiple objects can be contained and arbitrary precedence relations can be specified at any level of nesting.

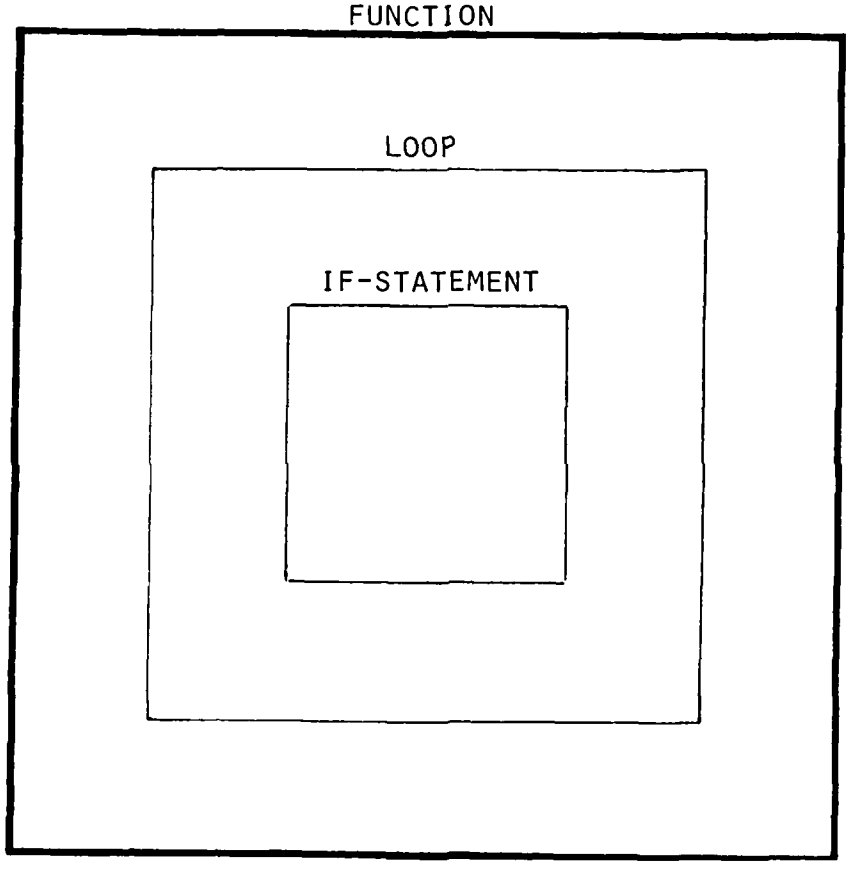


Figure 5: FIND ALL FUNCTIONS WHICH CONTAIN LOOPS THAT CONTAIN IF-STATEMENTS

Objects stored in the EPM have an explicit structure. For example, a function is composed of a *name*, *parameter-list*, and *body*. Such named parts of an object are called **slots** and may be used in creating the pictorial query. The

query "Find the FUNCTION named BAR" is shown in figure 6 as an example. In specifying the picture, the user asked to see the slot of the *function* box. By placing the string "BAR" in the sub-box labeled *name*, the user specifies that the string "BAR" must be contained in the *name* part of the function. When slots are not used, the containment will be considered satisfied if it occurs in any of the subparts of the outer object.

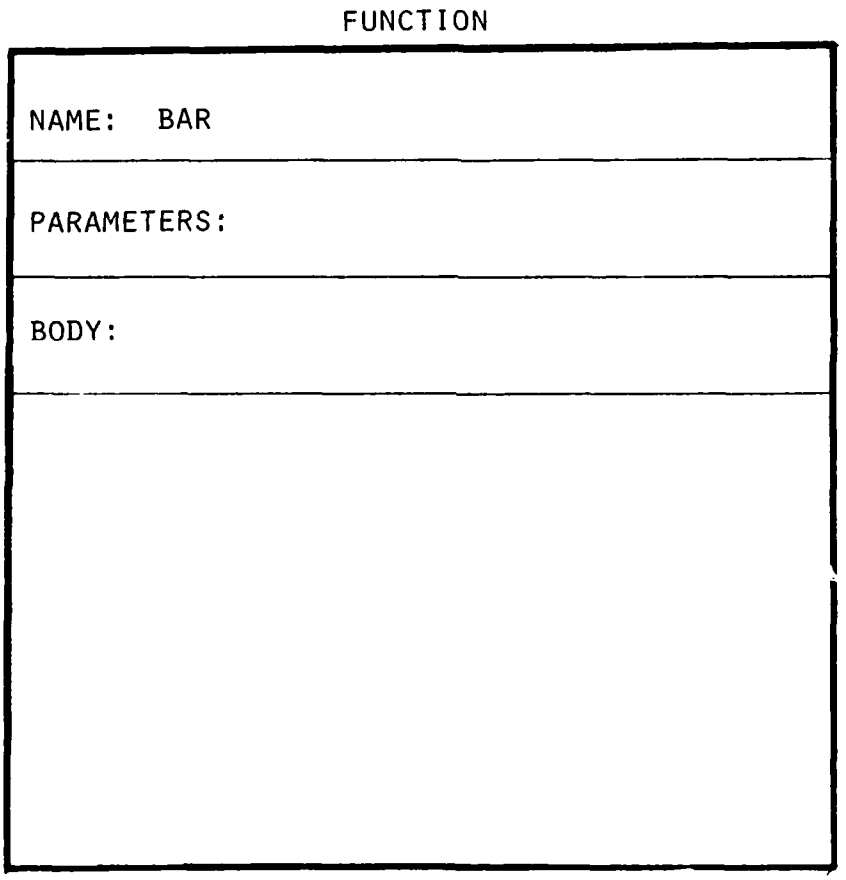


Figure 6: FIND THE FUNCTION NAMED BAR

Additional relations are introduced through the vocabulary of object types. For example, figure 7 illustrates the specification of the query "Find all FUNCTIONS that use the VARIABLE FOO." The box labeled *uses* represents a data flow object in the EPM database. It has a single slot which holds the object that is being "used." Similary, the relationship of one function calling another can be represented by the containment of a *calls* box, another valid EPM object, which in this case is shown from the control flow perspective. Since most important information about the program is represented explicitly in one of the views of the EPM, almost any important statement about a program can be made in terms of containment of objects.

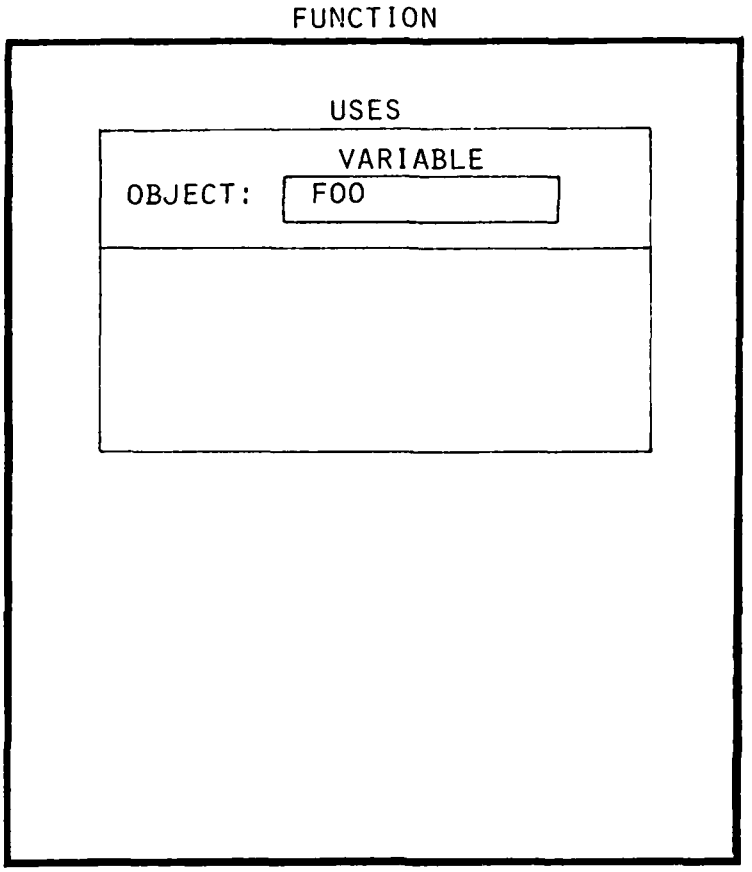


Figure 7: FIND ALL FUNCTIONS THAT USE THE VARIABLE FOO

A pictorial query can be composed of more than one top level box. All separate subpictures in the query that return some type of object must return the same type of object. The sets of that object type generated by the separate pictures are unioned together to form a single set as the answer to the entire query. Thus having separate top level boxes implies an *OR* operation, allowing the results from several variants of a simple query to be combined. For example figure 8 shows the picture for the query "Find all the FUNCTIONS that contain LOOPS or IF-STATEMENTS."

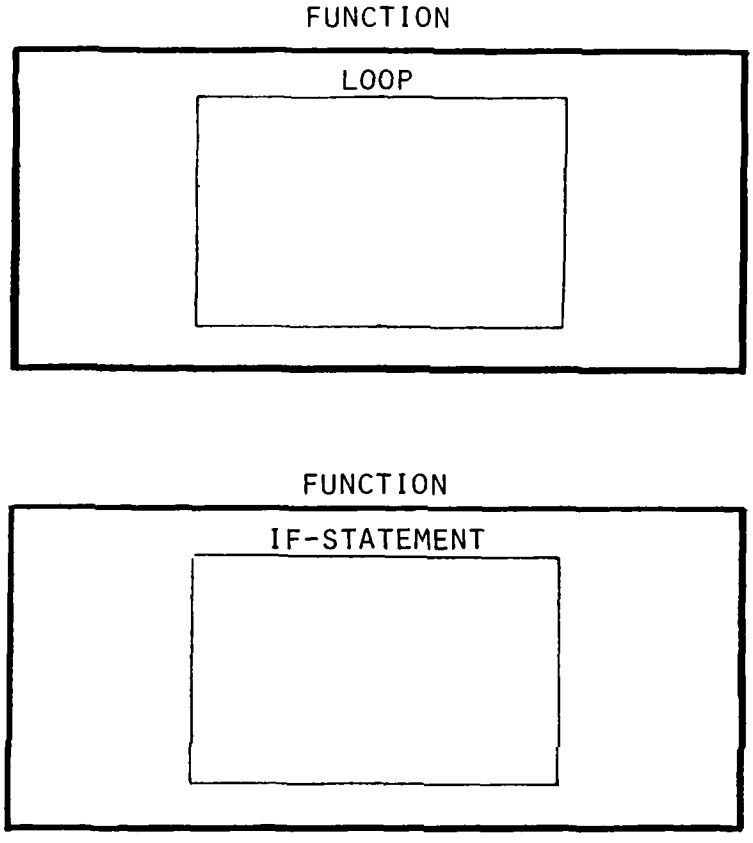


Figure 8: FIND ALL FUNCTIONS CONTAINING LOOPS OR IF-STATEMENTS

A special pair of boxes called *AND* and *OR* boxes are used to clarify representation of conjunctions and disjunctions. The boxes can be placed anywhere normal program object boxes are legal, but they have a special effect on the interpretation of the boxes they contain. Strictly speaking, these boxes are not essential, as any requests can be constructed without them. However, they are important from a user interface perspective (i.e., they make the PRL/PL easier to use), not from a mathematical perspective.

An *AND* box is satisfied if all the things it contains are satisfied. Figure 9 illustrates the use of an *AND* box to represent the query "Find the FUNCTIONS containing LOOPS and IF-STATEMENTS." Note that every normal program object box functions implicitly like an *AND* box as illustrated by the equivalence of figure 3 to figure 9. The primary use for *AND* boxes is where the default interpretation would be *OR*, such as within an *OR* box, or at the top level of the query.

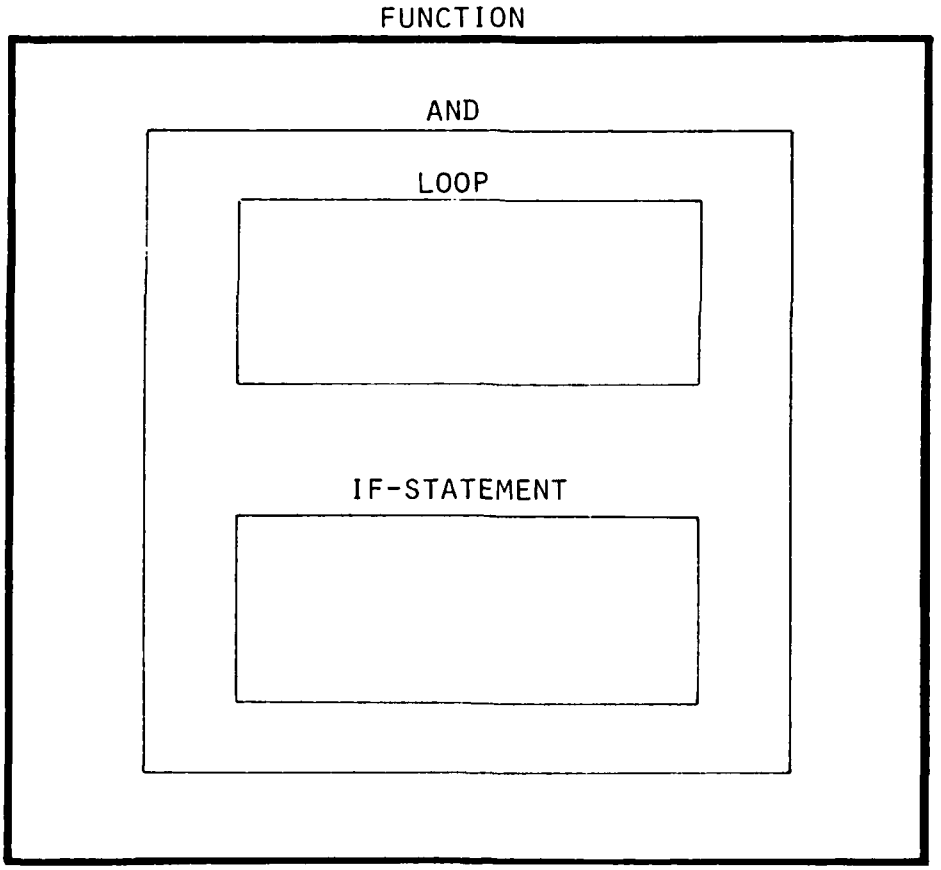


Figure 9: FIND ALL FUNCTIONS CONTAINING LOOPS AND IF-STATEMENTS

An *OR* box is satisfied if at least one of the things it contains is satisfied. Figure 10 illustrates the use of an *OR* box to represent the query "Find the FUNCTIONS containing LOOPS or IF-STATEMENTS." Note that in this case, the *OR* box is needed to force the desired interpretation, rather than that gained by the default of figure 3; it should be also noted that this is the same query as that represented in figure 8.

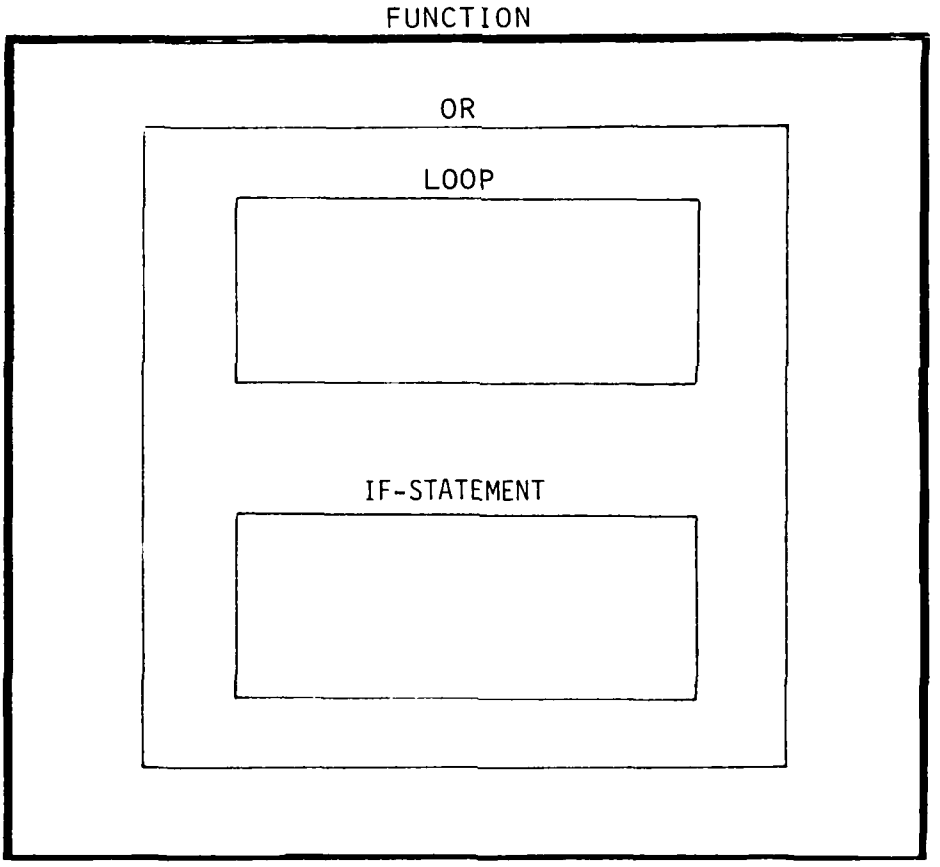


Figure 10: FIND ALL FUNCTIONS CONTAINING LOOPS OR IF-STATEMENTS

When constructing a search template in the PRL/PL, users may also specify objects whose appearance would invalidate the match. The notation for negation is to use a negative image (e.g., reverse video) in displaying the negated box. In the figures illustrated in this paper, slashes will be used to represent reverse video. For example figure 11 shows the picture representing the query, "Find the FUNCTIONS that do not contain LOOPS." It is also possible to negate parts of the context for an object which must appear. For example figure 12 shows the query, "Find the IF-STATEMENTS which are not contained in LOOPS."

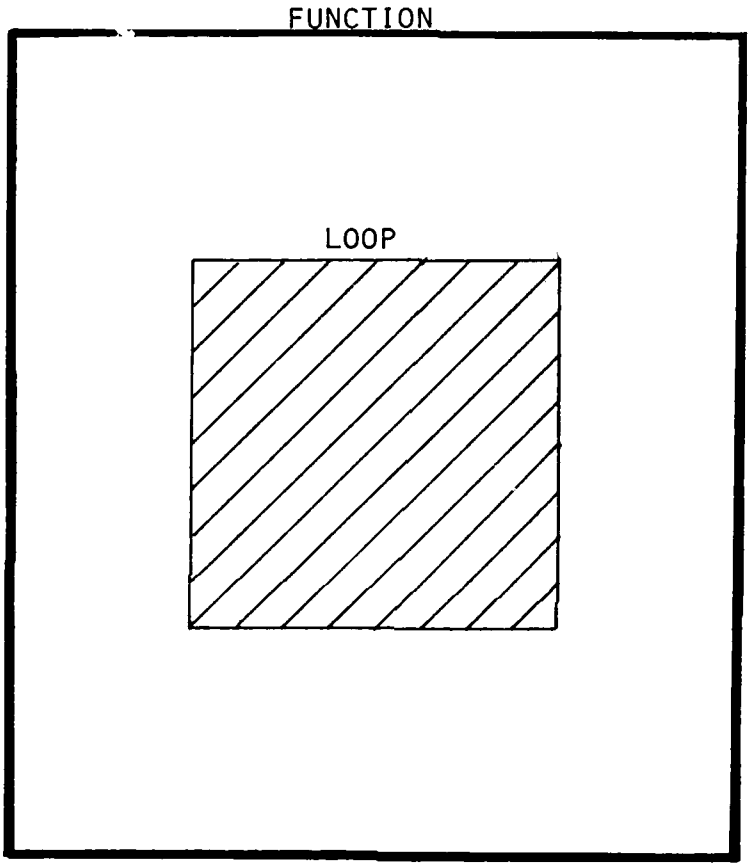


Figure 11: FIND THE FUNCTIONS THAT DO NOT CONTAIN LOOPS

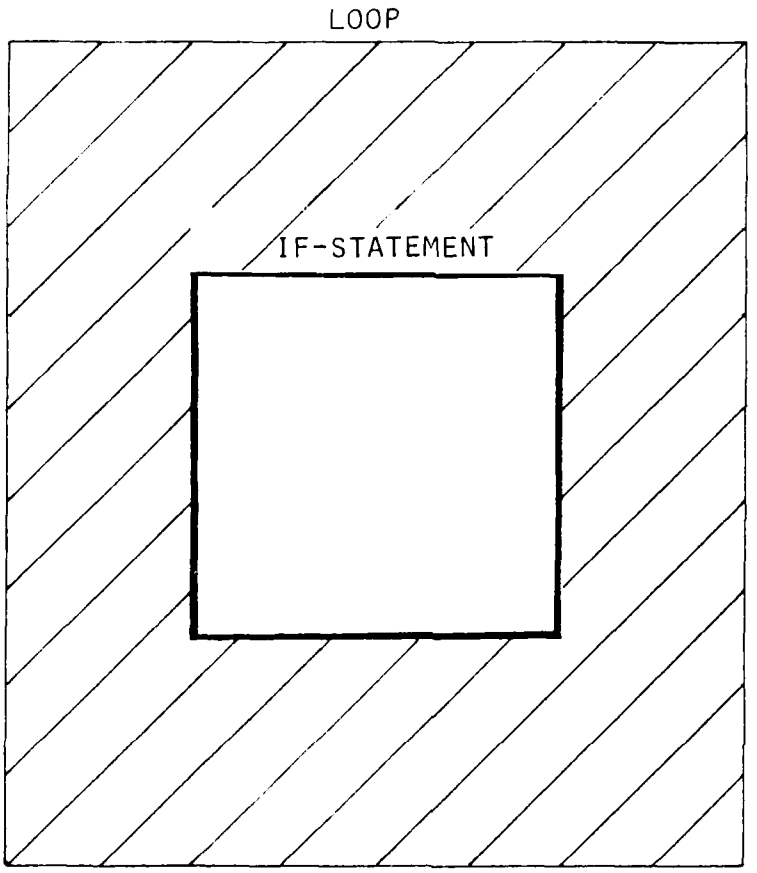


Figure 12: FIND THE IF-STATEMENTS NOT CONTAINED IN LOOPS

When used in combination with precedence arrows, a negated box will only disqualify a potential matching code fragment if the disallowed object appears in the specified relation to other objects. For example, figure 13 represents the query "Find the FUNCTIONS that have a LOOP not followed by an IF-STATEMENT." A matching *FUNCTION* may contain *IF-STATEMENTS*, as long as there is a *LOOP* which textually follows them.

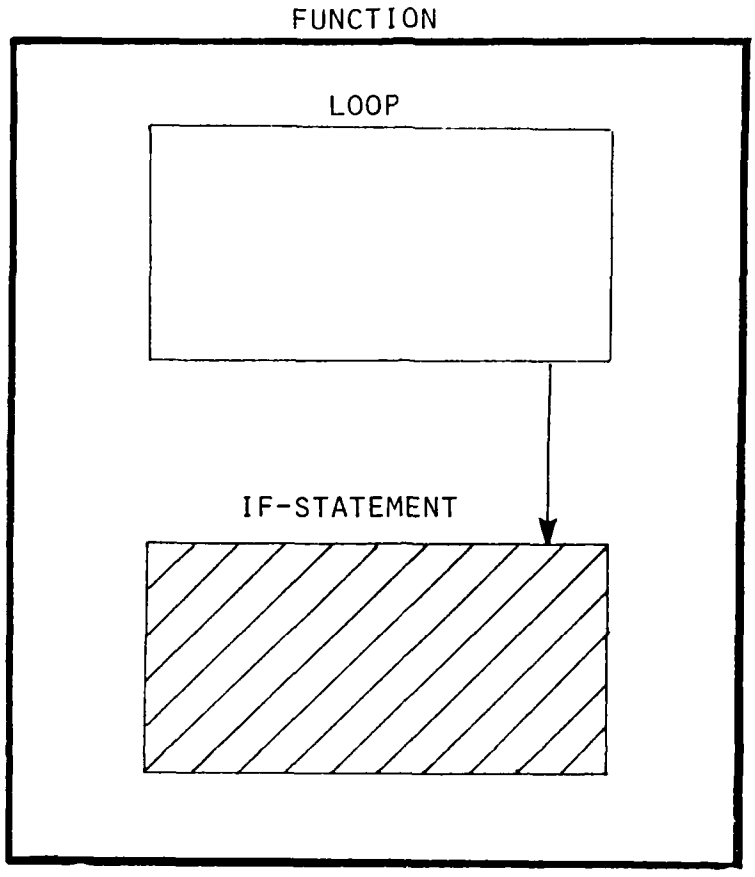


Figure 13: FIND THE FUNCTIONS WHICH HAVE A LOOP NOT FOLLOWED BY AN IF-STATEMENT

When a box representing a program object is instantiated, it is a statement that such an object must be present for a candidate region of code to be a successful match; all that is required is that one such object exist. Thus, all objects in PRL/PL queries are, by default, assumed to be existentially quantified.

There are other possible meanings a user might want to express. The PRL/PL allows the explicit specification of either universal quantification or

integer ranges for cardinality constraints on program objects. The operation of negation also has an implicit effect on the quantification of the negated terms.

Figure 14 depicts the query, "Find the FUNCTIONS in which all LOOPS contain IF-STATEMENTS." Compare this with figure 5; the two differ only in the explicit quantification on the box representing *LOOP*. The effect is that the mere existence of a *LOOP* containing an *IF-STATEMENT* is not enough to guarantee that a *FUNCTION* will pass the test. If there are other *LOOPS* contained in the *FUNCTION* which do not contain *IF-STATEMENTS*, then the *FUNCTION* will fail to satisfy the specified condition. Also note that it is not required that a *FUNCTION* contain any *LOOPS* to pass the test; if it has no *LOOPS*, then all the *LOOPS* it has contain *IF*-STATEMENTS. (Or, to put it another way, there is no *LOOP* which does not have an *IF-STATEMENT*.)

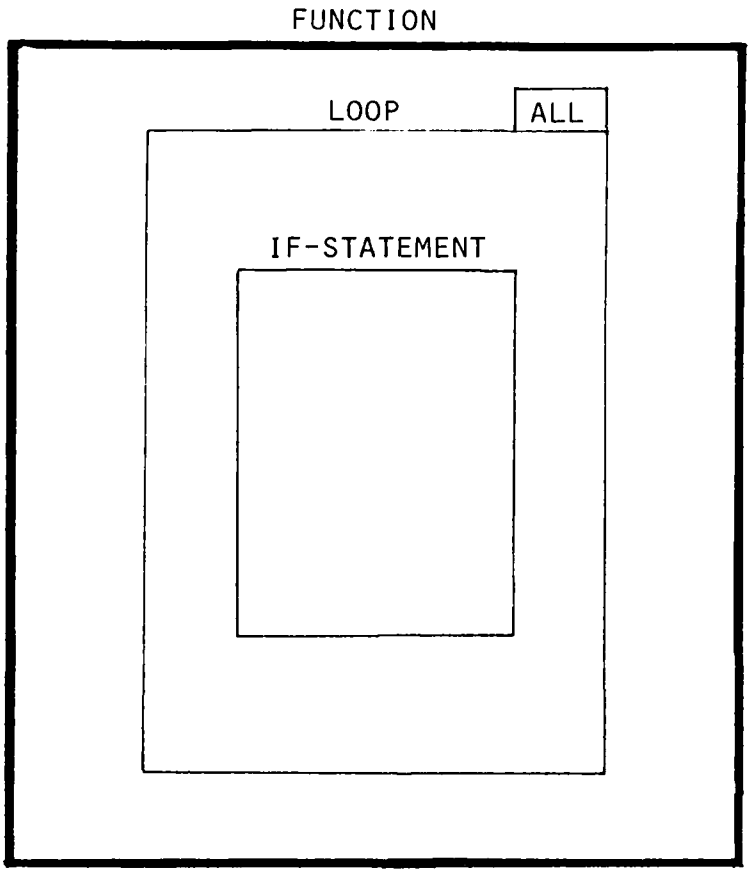


Figure 14: FIND THE FUNCTIONS IN WHICH ALL LOOPS CONTAIN IF-STATEMENTS

Figure 15 gives an example of a PRL/PL query that contains a cardinality constraint. A cardinality constraint can be any number of natural numbers or natural number ranges. The figure represents the query, "Find the FUNCTIONS which contain 2 LOOPS that contain IF-STATEMENTS."

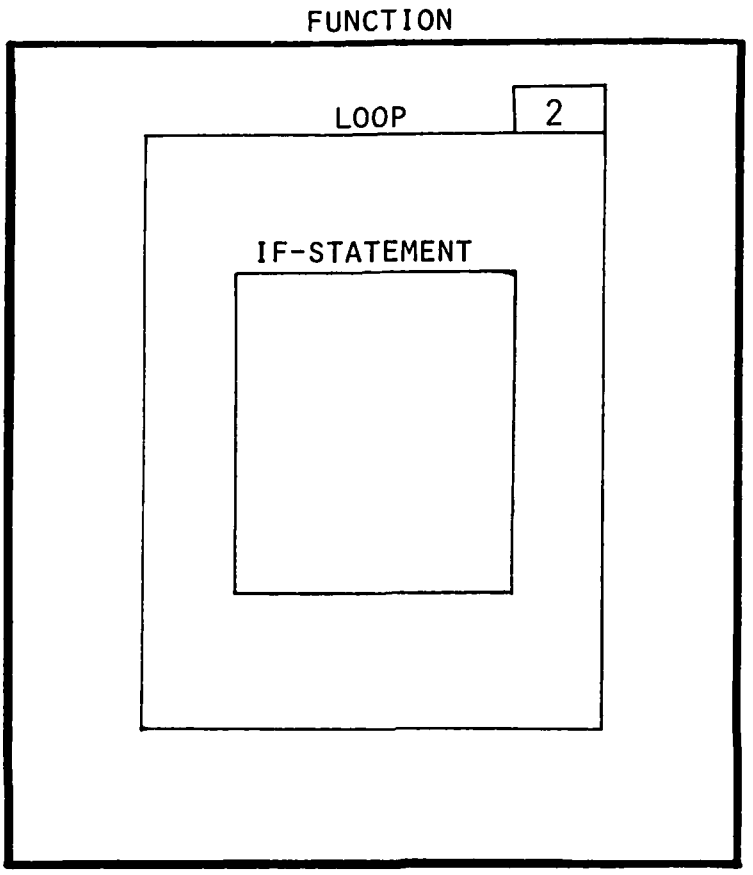


Figure 15: FIND THE FUNCTIONS WHICH CONTAIN 2 LOOPS THAT CONTAIN IF-STATEMENTS

Figure 16 shows an example with a negated box. The query means "Find the FUNCTIONS which contain an IF-STATEMENT not contained in a LOOP." Note that the quantification on the negated *LOOP* box has effectively been flipped to universal. The query means that for all the *LOOPS* in the *FUNCTION*, there is an *IF-STATEMENT* not contained in any of them. If the quantification had not been changed, the query would require the existence of some *LOOP* that did not contain an *IF-STATEMENT*, and other *LOOPS* in the

FUNCTION could still exist that did contain an *IF-STATEMENT.*

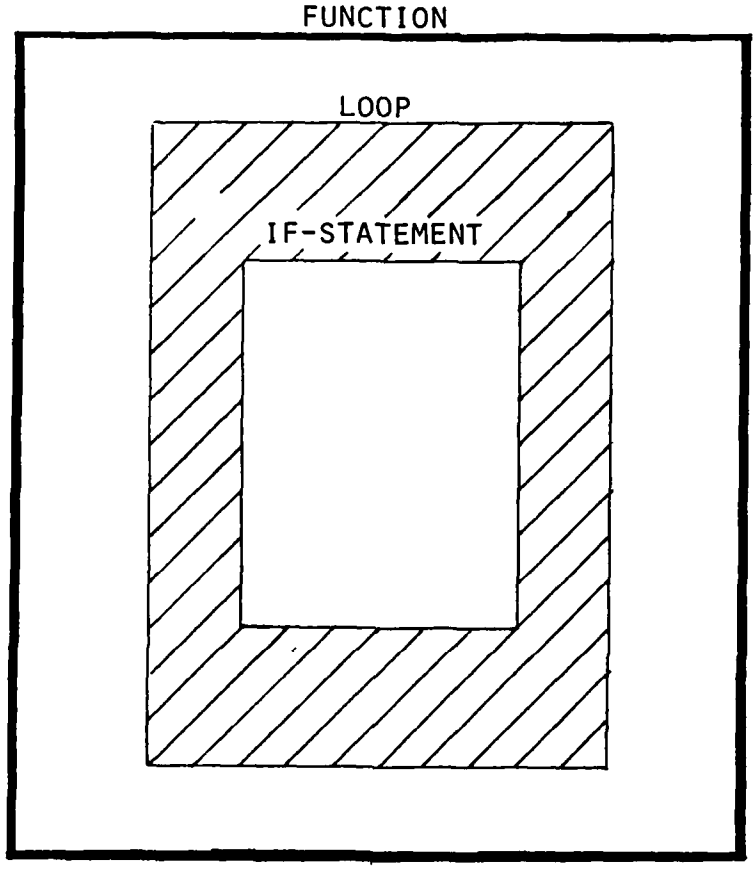


Figure 16: FIND THE FUNCTIONS WHICH CONTAIN AN IF-STATEMENT NOT CONTAINED IN A LOOP

The PRL/PL can handle query meta-variables. The notation for these variables is a character string surrounded by angle brackets. For example, <*foo*>, <*x*> and <*this-is-a-variable*> are all valid query meta-variables. Such variables may be contained in a box and are bound to a set of program objects of the type represented by that box. The effective binding of a query meta-variable is the intersection of the sets generated by its several uses in a query.

Meta-variables are generally introduced into a query to designate the same object as it appears in more than one context. For example, figure 17 illustrates the query, "Find all FUNCTIONS that USE a VARIABLE before SETTING that VARIABLE." Note that the meta-variable <*x*> appears in two places in the query, and is meant to refer to the same object. In this case, <*x*> represents a variable that is first used and then set.

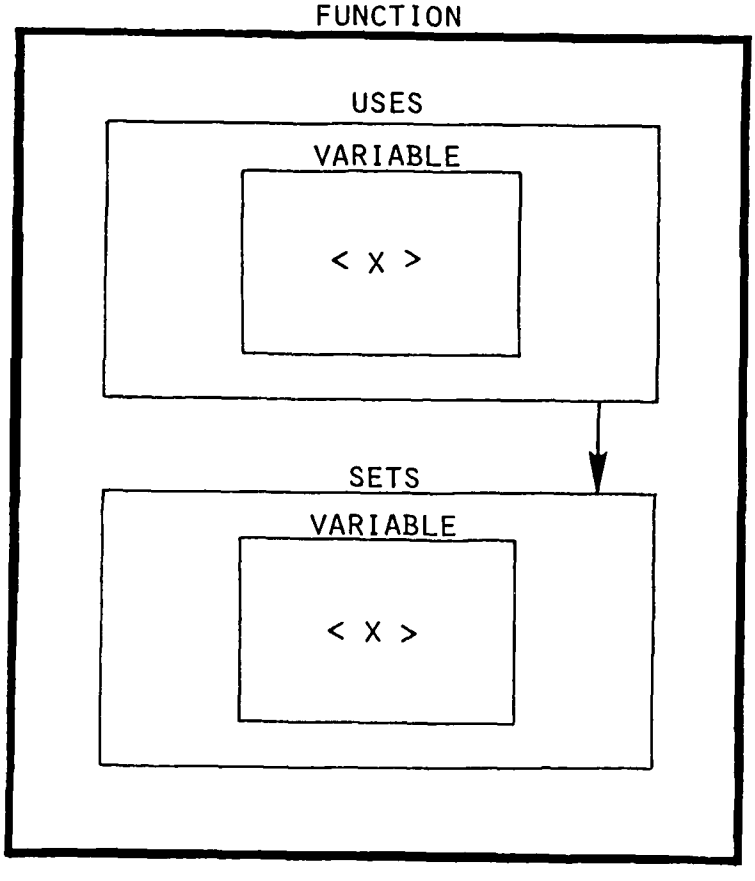


Figure 17: FIND THOSE FUNCTIONS THAT USE SOME VARIABLE BEFORE SETTING THAT VARIABLE

Actually, the query in figure 17 does not accomplish what was probably intended: checking for any use of a variable before it receives an initial value. Figure 18 accurately captures the intended meaning and illustrates the use of negation in combination with the precedence relation. Figure 18 translates as "Find the FUNCTIONS that do not SET a VARIABLE before USING that VARIABLE." Again note that <x> is used twice in the query and is intended to refer to the same variable name.

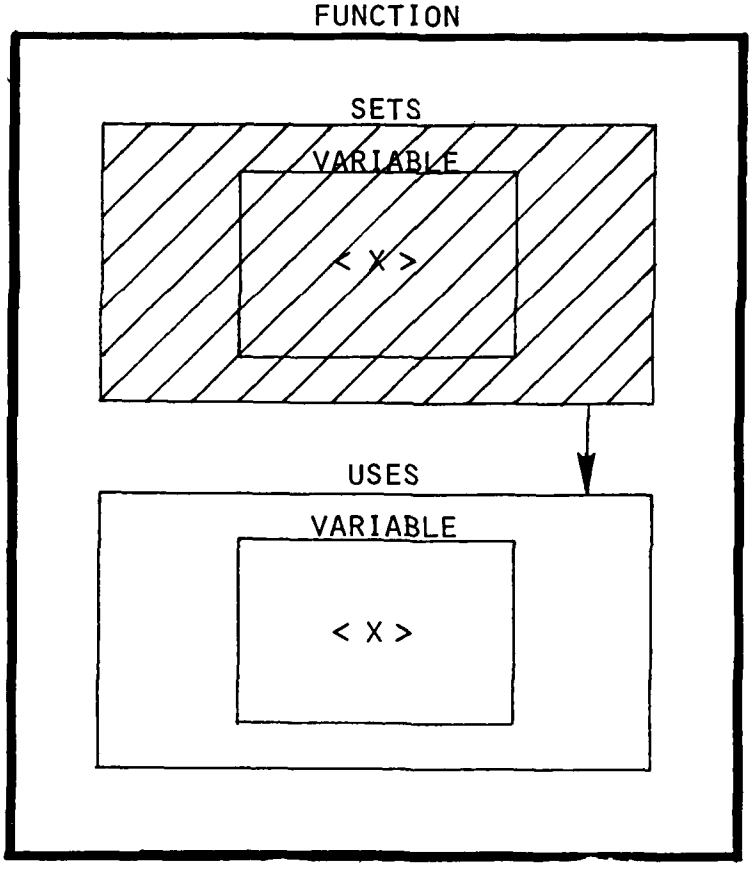


Figure 18: FIND THOSE FUNCTIONS IN WHICH A VARIABLE IS NOT SET BEFORE THAT VARIABLE IS USED

Figure 18 makes use of many of the features of the PRL/PL and still manages to maintain a high degree of comprehensibility. However, as illustrated by the example in figure 17, PRL/PL offers no protection against sloppy thinking.

3. FORMAL QUERY LANGUAGE: PRL/FL

The PRL Formal Language is intended to serve as an internal form for PRL queries. It should be both unambiguous and suitable for machine interpretation. It is possible that in the future there will be other external forms of the PRL in addition to the Picture Language. At such time, the Formal Language would serve as the common underlying representation. Since the PRL/PL may become cumbersome or counter-intuitive for some complex requests, the issue of a Natural Language interface may be considered in the future.

The current syntax of the PRL/FL is presented in BNF form in figure 19. This particular rendition is presented in prefix operator form and makes use of some rather long-winded keywords; it is still under development.

The least satisfactory aspect of this specification for the PRL/FL is its handling of precedence relations. The problem is that a precedence graph which is only constrained to be non-cyclic is transformed into a set of (possibly nested) binary relations. This may force some of the terms to be duplicated. We would like the PRL to keep track of the fact that such duplications are merely artifacts of the translation; it should create some sharable structure to represent the duplicated sub-queries.

QUERY	=	RETURN | CONTAINMENT | NEGATION DISJUNTION | CONJUNCTION
RETURN	=	(RETURN CONTAINMENT)
CONTAINMENT	=	(CONTAINS OBJECT CONTAINABLE)
NEGATION	=	(NOT CONTAINMENT)
DISJUNCTION	=	(OR QUERY*)
CONJUNCTION	=	(AND QUERY* ORDERING*)
ORDERING	=	(PRECEDES QUERY QUERY) (PRECEDES QUERY ORDERING) (PRECEDES ORDERING QUERY) (PRECEDES ORDERING ORDERING)
CONTAINABLE	=	NIL | STRING | VARIABLE | QUERY
OBJECT	=	(TYPE REPRESENTATION QUANTIZATION)
REPRESENTATION	=	TEXT | SYNTAX | VARIABLE | QUERY
QUANTIFICATION	=	EXISTENTIAL | UNIVERSAL | CARDINALITY
CARDINALITY	=	A SET OF NONNEGATIVE INTEGERS OR RANGES OF NONNEGATIVE INTEGERS
TYPE	=	ANY VALID TYPE OF PROGRAM OBJECT KNOWN TO THE EPM (CONSISTENT WITH THE STATED REPRESENTATION
NIL	=	AN EMPTY MARKER FOR BOXES WHICH ARE NOT CONSTRAINED TO CONTAIN ANYTHING
VARIABLE	=	< STRING >
STRING	=	OH COME ON!

Figure 19: PRL/FL: BNF SPECIFICATION

The open issues mentioned earlier in the discussion of the PRL/PL are really PRL/FL considerations. We know how we want the pictures to appear for all cases of combinations of quantification and negation. We know how they should be translated into the Formal Language. The open questions have to do with the interpretation of such queries. This is the province of the interpreter of the PRL/FL.

The mapping between the PRL/PL and PRL/FL is fairly straightforward. We already have a prototype unparser working which can take PRL/FL queries and draw the appropriate pictures. The parser, which will map from pictures to formal representation, will be part of the PRL/PL Editor Interface.

4. PRL/PL EDITOR INTERFACE

As important as the PRL/PL is the means by which a user enters and manipulates those queries. We are currently developing a prototype interface for the language. The interface is a graphical-style editor, with the characteristics of a syntax-oriented editor; i.e., it knows about the PL and ensures that requests are syntactically legal.

Given such a syntax editor for PRL/PL queries, the user can at all times see the whole evolving query as it is being composed and be certain that the query is syntactically valid. The query editor will take care of such layout issues as scaling the boxes appropriately and positioning the boxes when precedence relations are specified.

The prototype system is being developed on a Symbolics 3600 Lisp Machine, which has a high-resolution bit-mapped display terminal and mouse-input support. For each representation in the EPM, there will be a set of object types which may be instantiated as boxes and used in queries. As an alternative to typing in the name of the type of box to be instantiated, a complete catalog of types will be available on a set of mouse-sensitive menus. As a simple extension, the user may also maintain a library of previously constructed queries and query-fragments. The contents of this library will also be available on a mouse-sensitive menu. With this facility, commonly used queries or pieces of queries will always be easily available.

Most of the operations required to specify PRL/PL queries will benefit from the ability of a mouse to quickly designate a position on the screen. Placement of boxes in other boxes is natural with the mouse and easily transformable into the corresponding PRL/FL statement. Similarly, selecting a box for negation or as a return object is quickly accomplished and easily mapped to the formal representation. Specification of precedence relations works just as smoothly with a mouse.

Note that only boxes representing program objects will be displayed negated; *AND* and *OR* boxes when negated will be transformed into the equivalent positive statement through application of DeMorgan's Law. Also note that boxes which have been designated as return objects cannot be negated and that negated boxes cannot be selected as the query result.

5. PLANS FOR FURTHER DEVELOPMENT

5.1 QUESTIONS/ISSUES

There are several representation issues that still must be addressed. One of these involves the meaning of containment, which can be fuzzy and may vary with the types of objects. While it is clear what it means for one syntactic structure to contain another, it is less clear when considering more complex database structures. When searching for structures containing a cliche, the object may be distributed over several parts of a program. In this case, it may make sense to interpret containment only to require some overlap.

A problem in the PRL/PL involves the representation of multiple objects. When multiple objects are contained in some enclosing object, they can be connected into an arbitrary precedence graph by specifying directional arrows between any two boxes. Of course it is very easy to draw graphs that represent unsatisfiable conditions, as shown in figure 20. The system should be able to detect such configurations and bring them to the user's attention.

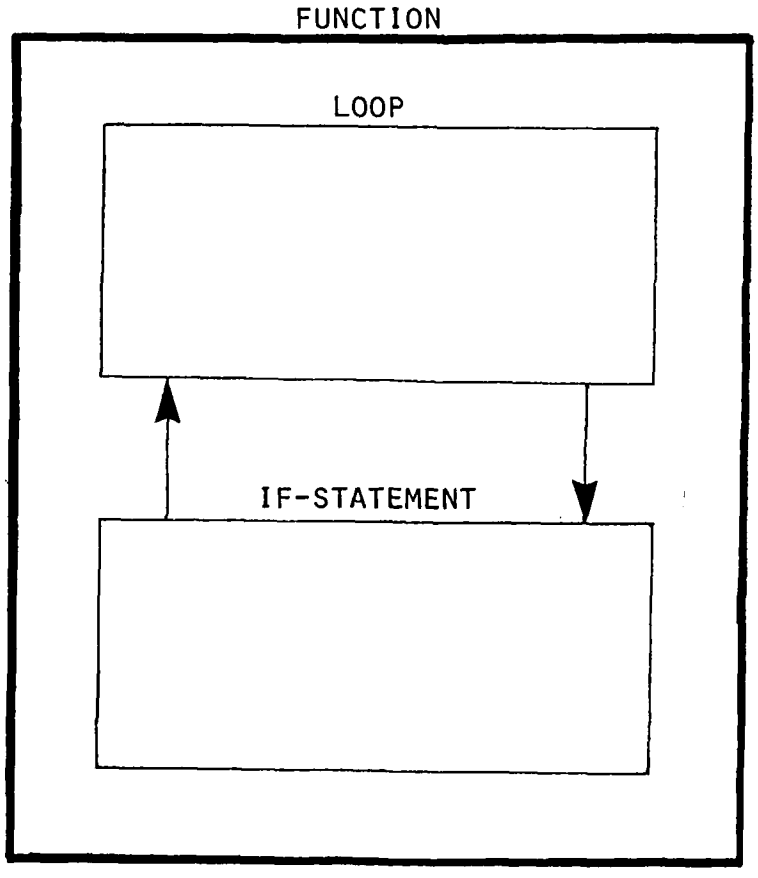


Figure 20: AN UNSATISFIABLE PRECEDENCE GRAPH

We are not satisfied that we have completely determined the correct interpretation for all possible combinations of quantifiers and cardinality. In particular, we do not yet feel comfortable with the interaction of these features with negation. On the whole, though, the default cases seem to work well to express useful queries. Improving our understanding of the underlying logic is one problem area to which we will be devoting more effort in the coming year.

5.2 FUTURE WORK

Part of our effort during the next year will be aimed at addressing the unresolved issues in the specification of the PRL/PL and the PRL/FL. Work will continue on the process of translating search requests in the PRL/FL to search requests in the EPM. In conjunction with this, we will begin looking at the issues of efficiently processing queries; part of this effort will be to refine the EPM search methods. An additional task will be to examine the problems involved in updating the internal database during a user's editing session. We will try to develop a strategy which allows for updates to the database only when necessary so that an unreasonable amount of time is not spent propagating changes. We will also continue the study of alternate interface forms.

6. PERSONNEL

6.1 PERSONNEL

The Program Reference Language (PRL) research project is being performed within the User Aids Program of AI&DS, with Dr. Brian P. McCune, Program Manager, as Principal Investigator. Other members of the AI&DS technical staff who have contributed to the project include Jeffrey S. Dean, Eric A. Domeshek, Michael A. Brzustowicz, Daniel G. Shapiro, and Susan G. Rosenbaum.

Dr. Brian P. McCune is the Principal Investigator of the PRL project. He received his Ph.D. in Computer Science from Stanford University in 1979; the title of his thesis was "Building Program Models Incrementally from Informal Descriptions." During the past decade, Dr. McCune has done research in the areas of artificial intelligence, software systems, and computer architecture, with emphasis on artificial intelligence approaches to software development and maintenance, information retrieval, database management, hypothesis formation, planning, and distributed processing. He has been the principal investigator of research projects to select and design candidate AI tools for assisting in the maintenance of Ada programs (sponsored by Rome Air Development Center), to design an intelligent program editor for Ada, to determine the feasibility of automatically generating operating systems, and to design and implement a knowledge-based system for textual information retrieval. Dr. McCune is on the

Editorial Advisory Boards of *Defense Electronics* and *The Artificial Intelligence Report.* He has been invited to discuss the application of artificial intelligence to defense problems numerous times, both at workshops and in published papers.

Jeffrey S. Dean is project leader of the PRL project. He is also currently leading the related Intelligent Program Editor project, and was previously the leader of the AI&DS Software Maintenance Project, which defined advanced Ada tools for software maintenance. He received his Masters degree in Computer Science/Computer Engineering from Stanford University, where he worked on the automatic derivation of operating systems. His main research interest is the application of AI to software tools. He came to AI&DS in January 1981 from Bell Telephone Laboratories, where he was involved in the development and maintenance of the UNIX operating system and its utilities.

Daniel G. Shapiro has been contributing to the PRL project since joining AI&DS in October 1981, after receiving a Masters degree in Electrical Engineering and Computer Science from the Massachusetts Institute of Technology. His research interests include artificial intelligence, expert systems, and software engineering. At AI&DS he has done work on expert systems for program and documentation editing, information retrieval, and mission planning. Currently, he is the leader of the Battlefield Commander's Assistant project, a basic research effort aimed at developing the AI technology required to assist battalion and/or brigade commanders in planning and evaluating tactics for combat situations. His masters thesis, entitled "Sniffer: A System that Understands Bugs," involved the design and implementation of a semantics-based debugger for the Programmer's Apprentice project at the MIT Artificial Intelligence Laboratory.

He also taught software engineering courses at MIT.

Eric A. Domeshek was responsible for much of the PRL experiment which studied how people think about programs and has played a key role in the development of the PRL Picture Language. Mr. Domeshek received an A.B. in Physics from Harvard College. His course work also emphasised computer science and cognitive science. His technical interests are in artificial intelligence, particularly knowledge representation, and computer graphics.

Michael A. Brzustowicz has been involved with the PRL project since joining AI&DS in November 1983. He received an S.B. degree in Physics from the Massachusetts Institute of Technology in 1979 and received his M.S.E.E. in Computer Engineering from Carnegie-Mellon University in 1980; his thesis work was entitled "A System for the Implementation of Models of Reasoning with Uncertain Data." Mr. Brzustowicz's current areas of interest include artificial intelligence, software engineering, ergonomic user interfaces, and computer-aided processes. Prior to joining AI&DS, Mr. Brzustowicz worked for the Development Systems Software Group of the Semiconductor Division of Texas Instruments, and for the Unix Development Group at Bell Laboratories.

Susan G. Rosenbaum has been working with the PRL project since joining AI&DS in June 1984. Her areas of interest include software engineering, artificial intelligence, and man-machine interfaces. She received a B.A. degree in Mathematics from the University of Texas at Austin in 1974 and an M.S. degree in Computer Science from the University of Texas at Arlington in 1979. Prior to joining AI&DS, she worked at Computer*Thought Corporation on the design and

development of a prototype tutoring system for the Ada language and at Texas Instruments in the Computer Science Laboratory.

6.2 INTERACTIONS

Dr. Brian P. McCune is an Associate Editor of *The AI Magazine*, the publication of the American Association for Artificial Intelligence. He is on the Editorial Advisory Board of *Defense Electronics* and also *The Artificial Intelligence Report.*

Dr. McCune presented a paper on an Intelligent Program Editor at a Software Engineering Technology Review sponsored by the Navy (July 1984). He was an invited speaker to COMPSAC '83 (November 1983) and EASCON '83 (September 1983), and was an invited participant to Knowledge Based Software Assistant Workshop at AAAI-83 (August 1983). He attended the NAVAIR/ONR Aviation Software Workshop (October 1983), the DARPA Formalized Software Development Workshop (November 1983), the Conference on Inference Theory and AI (November 1982), and the Software Maintenance Workshop (December 1983).

Dr. McCune attended the Eighth International Joint Conference on Artificial Intelligence (IJCAI-83), held in Karlsruhe, Germany, in August 1983 and the National Conference on Artificial Intelligence (AAAI-83), Washington D.C., August 1983.

Dr. McCune has been interfacing heavily with both operational and developmental commands in the Air Force and elsewhere in DoD and industry in order to understand current and future problems of software development and maintenance. Within the Air Force, Dr. McCune has met with personnel at the Air Force Office of Scientific Research, Rome Air Development Center, Wright Aeronautical Laboratories, Foreign Technology Division, Strategic Air Command headquarters, Air Force Communications Computer Programming Center, and Air Force Satellite Control Facility. Elsewhere in DoD he has talked with the Defense Intelligence Agency, Office of the Undersecretary of Defense for Research and Engineering, Defense Advanced Research Projects Agency, DoD STARS Program, Ada Joint Program Office, Office of Naval Research, Naval Electronics Systems Command, Naval Sea Systems Command, Naval Intelligence Command, Naval Research Laboratory, Naval Ocean Systems Center, Naval Intelligence Center, Naval Weapons Center, Army Research Office, Army Center for Tactical Computer Systems, and Army Ballistic Missile Defense Advanced Technology Center.

Dr. McCune has also visited numerous universities and research centers to assess the state of the art in automatic programming at first hand. Places visited include Harvard University, Massachusetts Institute of Technology, Carnegie-Mellon University, Duke University, University of California at Irvine, and Stanford University.

Jeffrey S. Dean presented a paper on an Automated Tool for Software Documentation at a Software Engineering Technology Review sponsored by the Navy (July 1984). He gave a paper on a study of software maintenance at the

Software Maintenance Workshop (December 1983). He attended the 7th International Conference on Software Engineering (March 1984); the Symposium for Application and Assessment of Automated Tools for Software Development (November 1983) and AAAI-83.

Daniel G. Shapiro was a panelist at the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, held in Pacific Grove, California, in March 1983. He presented papers on the PRL at the IEEE Trends and Applications Conference (May 1983) and the Seventh International Conference on Software Engineering (March 1984). He presented papers on information retrieval at AAAI-83 and IJCAI-83.

Eric A. Domeshek attended the Symposium for Application and Assessment of Automated Tools for Software Development (November 1983) and AAAI-83.

Michael A. Brzustowicz attended the Symposium for Application and Assessment of Automated Tools for Software Development (November 1983).

Susan G. Rosenbaum attended the ACM Conference on Lisp and Functional Programming (August 1982) and the National AdaTEC Conference (October 1983).

6.3 PUBLICATIONS

Members of PRL project staff have published a number of papers. A cumulative chronological list of publications appearing in technical journals and conference proceedings is listed below:

Thomas L. Adams, Andrew S. Cromarty, Brian P. McCune, Gerald A. Wilson, Milton R. Grinberg, James F. Cunningham, and Carl J. Tollander, "A Knowledge-Based System for Analyzing Radar Systems," invited paper, Proceedings, Military Microwaves '84, London, England, October 1984.

Daniel G. Shapiro, Jeffrey S. Dean, and Brian P. McCune, "A Knowledge Base for Supporting an Intelligent Program Editor," 7th International Conference on Software Engineering, March 1984. (See Appendix A.)

Andrew S. Cromarty, Daniel G. Shapiro and Michael R. Fehling, "Still Planners Run Deep: Shallow Reasoning for Fast Replanning," Proceedings, Society of Photo-Optical Instrumentation Engineers, Technical Symposium East, 1984, to appear.

Jeffrey S. Dean and Brian P. McCune, "An Informal Study of Software Maintenance Problems," Proceedings, Software Maintenance Workshop, December 1983. (See Appendix B.)

Brian P. McCune and Jeffrey S. Dean, "Trends for Advanced Software Tools," *Defense Science 2001+* (reprint of EASCON '83 paper), December 1983.

Brian P. McCune, Richard M. Tong, Jeffrey S. Dean, and Daniel G. Shapiro, "RUBRIC: A System for Rule-Based Information Retrieval," Proceedings, COMPSAC 1983, November 1983.

Brian P. McCune and Jeffrey S. Dean, "Trends for Advanced Software Tools," invited paper, Proceedings, EASCON '83, September 1983. (See Appendix C.)

Richard M. Tong, Daniel G. Shapiro, Brian P. McCune, and Jeffrey S. Dean, "A Rule-Based Approach to Information Retrieval: Some Results and Comments," Proceedings, National Conference on Artificial Intelligence, Washington, D.C., August 1983.

Richard M. Tong, Daniel G. Shapiro, Jeffrey S. Dean, and Brian P. McCune, "A Comparison of Uncertainty Calculi in an Expert System for Information Retrieval," Eighth International Joint Conference on Artificial Intelligence, Karlsruhe, West Germany, August 1983.

Brian P. McCune and Robert J. Drazovich, "Radar with Sight and Knowledge," invited paper, *Defense Electronics*, August 1983.

Richard M. Tong and Daniel G. Shapiro, "An Experiment with Multiple Valued Logics in an Expert System," Proceedings of the IFAC Symposium on Fuzzy Information, Knowledge Representation and Decision Analysis, Marseille, France, July 1983.

Daniel G. Shapiro and Brian P. McCune, "The Intelligent Program Editor: A Knowledge-Based System for Supporting Program and Documentation Maintenance," Proceedings of the Trends and Applications Conference of the IEEE, May 1983.

Gerald Wilson, Eric A. Domeshek, Ellen L. Drascher, and Jeffrey S. Dean, "The Multipurpose Presentation System," Proceedings, Very Large Data Base Conference, 1983.

Jeffrey S. Dean and Brian P. McCune, "Advanced Tools for Software Maintenance", Rome Air Development Center, RADC-TR-82-313, December 1982.

Brian P. McCune, Jeffrey S. Dean, Daniel G. Shapiro, and Richard M. Tong, "Rule-Based Information Retrieval," *Workshop on Intelligence Applications of Advanced Computer and Information Technology: Focus on Artificial Intelligence*, Office of Research and Development, Office of Scientific and Weapons Research, Central Intelligence Agency, Washington, D.C., November 1982.

Robert J. Drazovich, Brian P. McCune, and J. Roland Payne, "Artificial Intelligence: An Emerging Military Technology," invited paper, *Conference Record, EASCON '82: Fifteenth Annual Electronics and Aerospace Systems Conference*, Institute of Electrical and Electronics Engineers, Inc., Washington, D.C., September 1982, Pages 341-348.

Brian P. McCune, editor, "AI at AI&DS," *The AI Magazine*, Volume 2, Number 2, Summer 1981, pages 44-47.

Daniel G. Shapiro, "Sniffer: A System that Understands Bugs," MIT/AIM/638, June 1981.

Brian P. McCune, "Incremental, Informal Program Acquisition," *Proceedings of the First Annual National Conference on Artificial Intelligence*, Stanford University, Stanford, California, August 1980, pages 71-73.

Daniel G. Shapiro, "A Proposal for Sniffer, A System that Understands Bugs," MIT/AI Working Paper 202, July 1980.

Cordell Green, Richard P. Gabriel, Elaine Kant, Beverly I. Kedzierski, Brian P. McCune, Jorge V. Phillips, Steve T. Tappel, and Stephen J. Westfold, "Results in Knowledge-Based Program Synthesis," *IJCAI-79: Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, Volume 1, Computer Science Department, Stanford University, Stanford, California, August 1979, pages 342-344.

George R. Lewis, J. Shirley Henry, and Brian P. McCune, "The BTI 8000: Homogeneous, General-Purpose Multiprocessing," in Richard E. Merwin, editor, 1979 National Computer Conference, *AFIPS Conference Proceedings*, Volume 48, AFIPS Press, Montvale, New Jersey, June 1979, pages 513-528.

Cordell Green and Brian P. McCune, "Knowledge-Based Programming Applications," *Applications of Image Understanding and Spatial Processing to Radar Signals for Automatic Ship Classification: Proceedings of a Workshop*, Naval Electronic Systems Command, Washington, D.C., February 1979, pages 94-99.

Cordell Green and Brian P. McCune, "Application of Knowledge-Based Programming to Signal Understanding Systems," *Distributed Sensor Nets: Proceedings of a Workshop*, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, December 1978, pages 115-118.

Brian P. McCune, "The PSI Program Model Builder: Synthesis of Very High-Level Programs," Proceedings of the Symposium on Artificial Intelligence and Programming Languages, *SIGPLAN Notices*, Volume 12, Number 8, *SIGART Newsletter*, Number 64, August 1977, pages 130-139.

7. REFERENCES

1. Domeshek, Eric A., Shapiro, Daniel G., Dean, Jeffrey S., McCune, Brian P., "An Informal Study of Program Comprehension", AI&DS TM-1014-3, March, 1984.

2. Shapiro, Daniel G., "Sniffer: A System that Understands Bugs", AIM-638, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Mass., 1981.

3. Shapiro, Daniel G., McCune, Brian P., "Searching a Knowledge Base of Programs and Documentation", AI&DS TM-1014-2, January 1983.

4. Shapiro, Daniel G., McCune, Brian P., "A Knowledge Based System for Supporting Program and Documentation Maintenance", Proceedings, IEEE Trends and Applications, 1983, pp. 226-232.

5. Shapiro, Daniel G., Dean, Jeffrey S., and McCune, Brian P., "A Knowledge Base for Supporting an Intelligent Program Editor", Proceedings, 7th International Conference on Software Engineering, 1984.

6. Zloof, M. M., "System for Business Automation," *Communications of the ACM*, 1977. Vol 20, No. 6. pp. 385-396.

APPENDIX A.

This appendix contains a reprint of the paper "A Knowledge Base for Supporting an Intelligent Program Editor," by Daniel G. Shapiro, Jeffrey S. Dean, and Brian P. McCune.

[Seventh International Conference on Software Engineering, March 1984.]

A KNOWLEDGE BASE FOR SUPPORTING AN INTELLIGENT PROGRAM EDITOR

Daniel G. Shapiro
Jeffrey S. Dean
Brian P. McCune

Advanced Information & Decision Systems
201 San Antonio Circle
Mountain View, CA 94040

ABSTRACT

This paper presents work in progress towards a program development and maintenance aid called the Intelligent Program Editor (IPE), which applies artificial intelligence techniques to the task of manipulating and analyzing programs. The IPE is a knowledge based tool: it gains its power by explicitly representing textual, syntactic, and many of the semantic (meaning related) and pragmatic (application oriented) structures in programs. To demonstrate this approach, we implement a subset of this knowledge base, and a search mechanism called the Program Reference Language (PRL), which is able to locate portions of programs based on a description provided by a user.

This research was supported by the Air Force Office of Scientific Research under contract F49620-81-C-0067, the Office of Naval Research under contract N00014-82-C-0119, and Rome Air Development Center under contract F30602-80-C-0176.

1. INTRODUCTION

The effort and expense involved in software maintenance have been recognized as a major limitation on the capabilities of current software systems. In a study on software maintenance issues in the Air Force, we found that the process of comprehending the form and function of existing software (i.e., what it does and how it does it) is the largest task in the maintenance process [2].

The basic cause of this "comprehension problem" is the loss of knowledge during the programming process, caused by factors such as poorly written software, inadequate documentation, programmer forgetfulness, and personnel turnover. To address these issues, we have started a project to develop intelligent, knowledge-based programming aids, designed to help the programmer overcome limitations of more traditional tools. This paper describes the initial phase of one of these tools, an editor known as the Intelligent Program Editor (IPE). The following sections discuss the motivation behind intelligent editing, the design of an intelligent editor, a database for the editor, and a scenario demonstrating an actual implementation of a portion of the IPE's database, used in the context of a program search.

2. MOTIVATION

An intelligent editing system is a sophisticated tool for developing and maintaining programs. The goal, insofar as it is possible, is to decrease the amount of information a programmer needs to supply in order to create and maintain a program, and to simultaneously increase the reliability of the resulting code. This can be accomplished by incorporating knowledge about the structure and intention of programs into the editing tools used to develop and maintain them. Perhaps the best way to illustrate this approach is to present an allegory having to do with the production of a technical manuscript.

Assume that there is a manuscript which needs to be typed for publication. If it is given to a typist who does not speak English, the result would be, at best, a word-for-word copy of the original manuscript. If it is given to an English-speaking typist, simple errors, such as misspellings and punctuation problems, might be fixed during the typing process. If the manuscript is given to an English teacher moonlighting as a typist, the result might well be a version in which the prose is smoothed and otherwise improved. Finally, if one is lucky enough to find a typist familiar with the domain of discourse (such as the author), the resulting document might even have factual errors corrected and incomplete thoughts identified.

A programmer selecting an editor system for writing code is in a similar situation. A standard text editor is comparable to the non-English-speaking typist; text appears exactly as it is typed, with no enhancements. The English-speaking typist could be compared to a syntax-oriented editor, which can eliminate syntactic program errors and misspelled keywords. The English teacher/typist knows about the language itself but not about the content of the thoughts. This situation is comparable to a programming language-specific editor which applies knowledge about the domain of programming; this editor can instantiate general programming techniques, catch certain types of semantic errors, make style suggestions, and improve the overall flow of the program. The technical typist who understands the content of what is being said is analogous to an editor that

utilizes knowledge about the application domain; it can help in algorithm development and can catch certain types of pragmatic errors which are dependent upon the specific application domain.

3. THE INTELLIGENT PROGRAM EDITOR

The Intelligent Program Editor (IPE) described in this paper most closely corresponds to the English teacher/typist mentioned above, in that it will support textual and syntactic manipulations, and have the ability to assist in the implementation of typical programming actions. This power is obtained through the use of a database that explicitly represents the functional organization of programs in terms of textual, syntactic, and intention-oriented structures. With this database, the IPE is in a position to become more of a programming environment than solely an editing tool. In this vein, we are interested in addressing the following design goals [5].

The IPE should provide a means for naturally incorporating documentation into the program development process. In our view, this requires the ability to link documentation into the organizational structure of a program (similar to Nelson's [3] concept of Hypertext), and the ability to actively use any information that is supplied (to provide programmers with a motivation for including descriptive data). In the IPE, documentation will provide input to a program search facility.

The system should support incremental program analysis. The object here is to employ the system's understanding of program structure to catch syntactic and certain semantic errors prior to execution. Examples include identifying variables that are accessed before being set (via data flow analysis) and detecting programming cliches that have been incompletely implemented. There is also a role for error prevention: some editors (e.g., [6]) prevent syntactic errors from ever occurring.

The IPE will allow the user to employ alternate program visualizations. This means allowing the programmer to examine or modify code through any of the representations mentioned above. For example, a syntax based approach might be appropriate during program construction, while a graphical data flow display may be useful within the debugging process.

All of these capabilities require the use of multiple program representations, as well as mechanisms for searching and manipulating the information they contain. Therefore, in the first phase of the IPE project, we constructed a prototype version of this program database, called the Extended Program Model (EPM), and demonstrated it in the context of program search. The remainder of this paper discusses the EPM and the search example that was produced.

4. THE EXTENDED PROGRAM MODEL

The Extended Program Model (EPM) provides a new way of representing and accessing programs by defining a vocabulary for discussing programs which uses terms that are much closer to the ones which users naturally employ. The EPM provides this capability through the use of a database that represents the structure of programs from a variety of views. The EPM can form the backbone for a number of systems which exhibit a deep understanding of the organizational structure and meaning of code.

The EPM is constructed in terms of two major subsystems (see Figure 1) : a program structures database and a search and update component called the Program Reference Language, which provides access to the database. In addition, the EPM will contain a library of "rational form" constraints that will monitor program composition for its structure and intentional content. As a whole, the system can be thought of as a database management system for creating and maintaining code. It provides a search language for accessing its knowledge, a facility for performing updates, as well as a set of semantic integrity and consistency constraints for monitoring the validity of the data it contains.

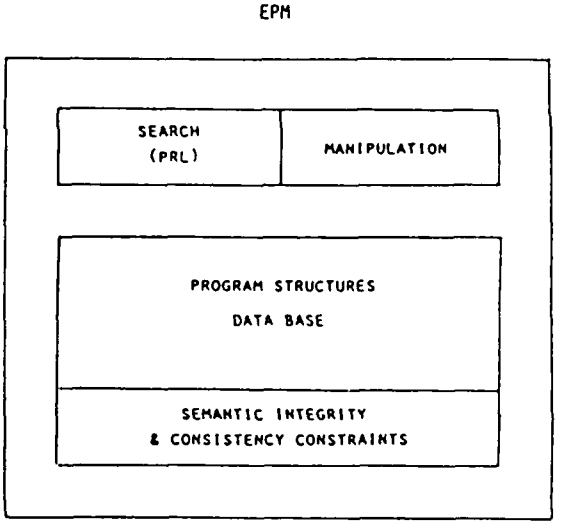


Figure 1. The Extended Program Model

4.1 THE PROGRAM STRUCTURES DATA BASE

The EPM's program structures database is constructed in terms of a collection of representations which reflect the transition from a syntactic to a more intention-oriented analysis of code (Figure 2). We are considering these viewpoints to be abstract data types which facilitate different sorts of retrieval operations.

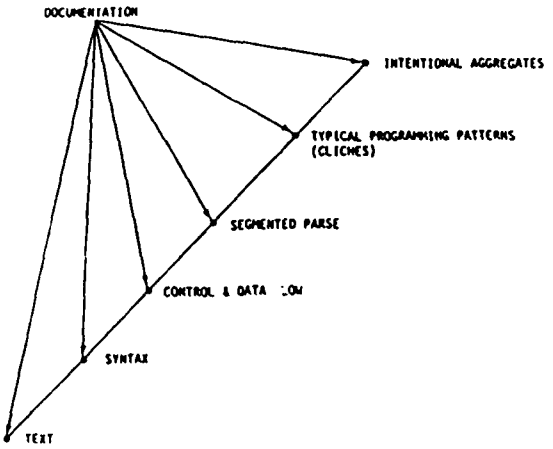


Figure 2. Representation Levels in the EPM

The textual representation gives the EPM the view that most text editors provide. It is a low-level approach, concerned with words and delimiters, but it allows for important textual search operations.

The syntactic viewpoint embodies the rules of grammar for particular programming languages. The syntactic database provides the EPM with a vocabulary for programming constructs such as "for" loops, parameters, and procedures.

The next level of representation is the flow level, which provides standard data and control flow information. It provides a vocabulary relating to the logical structure of programs.

The segmented parse representation defines a vocabulary for a program in terms of its component data and control flow. For example, iterations are decomposed into a set of roles which identify the subfunctions of a loop. In the breakdown we are using, loops contain generators, filters, terminators, and augmentations [7]. Generators are segments which produce a sequence of values. They can be further refined into initializations and a body, which is the portion that is executed many times. Filters restrict that sequence of values. A terminator is like a filter, except that it has the additional potential to stop execution of the loop. An augmentation consumes values and produces results. There are other vocabulary elements for describing straight line code.

The taxonomy discussed up to this point primarily captures information about the form of programs (as opposed to their meaning). The only semantic elements we have introduced describe the substructure of built-in entities such as loops. The next (more abstract) viewpoint considers programs to be built of objects with stereotyped purposes. These are called typical programming patterns (TPPs). Examples of TPPs include variable interchanges, list insertions, and hash table abstractions. These abstractions are the tools employed by every expert programmer. Rich has defined a library of such TPPs [4] (he uses the term cliche; in this paper, we use both terms interchangeably).

The remaining databases (intentional aggregates and documentation) provide methods for associating the intentions behind a program with specific features of code. They capture pragmatic knowledge relating to the domain of application of the program. Intentional aggregates are a type of formal documentation that allow the association of larger program fragments with key concepts (supplied by the user). They can be used to collect a set of TPPs and other program segments that implement a single conceptual function; for example, a collection of TPPs representing queue operations might be grouped (by the user) into an intentional aggregate representing a scheduler.

The documentation database allows the user to associate comments with any of the program features already described. At the lowest (i.e., textual) level, this would take the form of in-line comments. At other representational levels, the user could, for example, document the data flow in a particular module (saying why an input-output relationship occurs), justify his use of particular TPPs, or explain why particular syntactic features are employed. The advantage of this technique over current documentation practice is the ability to make a direct association (via links maintained by the IPE) between the documentation and what it talks about, at an appropriate conceptual level.

4.2 KNOWLEDGE ACQUISITION

Since the EPM's database is intended to support an actual editing system in the near future, it is important to address the question of where its information is obtained. In our approach, the different knowledge sources are acquired in part from the user, and in part by automatic means. Specifically, the syntactic representation can be obtained directly from the textual representation, and the segmented parse viewpoint can be constructed through data flow analysis techniques of the kind developed by Waters [7].

The TPP structures are harder to obtain. Recent research efforts indicate that general recognition of cliches may be possible [1], but at the current time, these techniques have not actually been demonstrated. The EPM will use manual recognition techniques (at least until automatic recognition techniques have been refined). There are two manual recognition techniques planned for the system. In the first, the user points to a piece of code and identifies it as being a particular TPP (as a way of documenting the system); at this point, once the scope has been narrowed down, it may be possible to identify the subcomponents of these programming cliches automatically. In the second method, the user uses TPPs for program generation (as in [8]); by instantiating a TPP and "filling in the blanks," the EPM can acquire all the necessary information.

The intentional aggregate and documentation views must be wholly obtained from the user. At a minimum, the EPM's planned consistency mechanisms will identify any of this information that may be out of date due to modifications to the code.

5. THE PROGRAM REFERENCE LANGUAGE

In order to demonstrate the feasibility of the EPM, we implemented a portion of the database described above, and built a version of the EPM's search facility, the Program Reference Language (PRL) which operates on that data. The PRL is a tool for locating regions of program text based upon a description provided by the user. As a support system, it provides programmers with an intention-oriented vocabulary for specifying portions of programs in situations where they may be unfamiliar with the detailed structure of the code. This might occur in the process of editing programs which may be too large to remember explicitly, or in the act of understanding code which has rarely been seen before (as is often the case in maintenance).

The PRL demonstration system allows program search based on four of the representations described above, namely the textual, syntactic, segmented parse and typical programming pattern views (Figure 3). These databases are connected through a code region abstraction that associates program features with physical sections of program text.

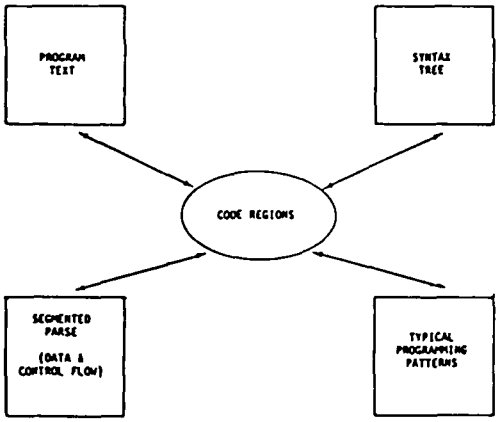


Figure 3. The Program Reference Language Implementation

The PRL has a flat information structure. It represents each database in terms of a complex tree or graph structure of frames. Although the system can arbitrarily convert between viewpoints by using code regions as an intermediary, the databases have no direct links between one another. These conversions are inherently heuristic since the separate representations do not necessarily have a one-to-one correspondence. The information in each database is either automatically derived, or can be reasonably obtained from the user. In situations where the latter is necessary, we have assumed that information may be provided in an incomplete form.

5.1 CODE PAINTING

From a computational point of view, the main problem involved with this multiple representation approach is to define a mechanism that is able to compare information obtained from the different sources of knowledge. The PRL accomplishes this via the code region abstraction, which functions as a common language that each of the representations can use to communicate.

Code regions support two different approaches to search. In the first method, which we call sequential filtering, the user makes a gross stab at selecting a code region by generating all of the elements which satisfy some fairly general condition. He then sequentially restricts that set by applying more and more conditions. For example, to find "the loop which computes the sum of the test scores", he locates the set of all loops, and then restricts it to the ones which involve test scores and summations.

In the second approach, the user identifies a collection of items, possibly from several different databases, and intersects them together to find the elements which satisfy all of the conditions he wants to impose. In this "code painting" approach, the PRL combines these items essentially by overlaying the corresponding regions of code. For example, locating "the loops which compute sums" is done (figuratively) by coloring all loops red and all places that compute sums yellow. Any region which comes up orange has all of the properties that were desired.

Code painting is a deliberately coarse affair. It is designed to exploit the kind of incomplete or even slightly inaccurate information which the EPM will contain, given that much of the data is provided by the user. In some cases, code painting may not identify the exact section of the program which the user desired, but in the context of an interactive system with a screen oriented display, "close" will be good enough. To help the user see the effects of code painting, it is possible to highlight the identified section(s).

5.2 A SCENARIO USING THE PRL

The following example shows how the PRL uses the code painting paradigm to answer the question "find the initializations of the loop which computes the sum of the test scores", given the Ada program shown in Figure 4.

```
for MAXSIZE in 1..10 loop
  TOTAL:= ARRAYSUM (TEST-SCORES, MAXSIZE);
  put (TOTAL);
end loop;


function ARRAYSUM (A: in ARRAY; N: in INTEGER) return INTEGER is
  begin
    SUM: REAL:= 0;
    for I in 1..N loop
      SUM:= SUM + A(I);
    end loop;
    return SUM;
end ARRAYSUM;
```

Figure 4. The Ada Program Used in the Scenario.

In this example, the user starts by identifying three sets of data, corresponding to the summation TPPs, syntactic loops, and segmented parse frames involving the test score array.

```
> (index 'summation tpp-database)
   => TPPset1

> (index 'loops syntax-database)
   => LOOPset1:[length 2]

> (index 'TEST-SCORES segp-database)
   => SEGset1:[length 6]
```

The program only contains one TPP, but there are two loops, and several segments which relate to the variable TEST-SCORES. It is important to notice that all of these segments use the data contained in the variable TEST-SCORES but do not necessarily refer to it by that name (for example, the literal "A(I)" in the ARRAYSUM function accesses the test score array). This association is apparent from the data flow analysis within the segmented parse.

The user intersects these descriptions by invoking the code painting paradigm. The code-painting algorithm returns the largest region of text which can be described in all three ways.

```
> (overlay-code-regions TPPset1 LOOPset1 SEGset1)
   => CODE-REGION1
      **for I in 1..N loop
           SUM:= SUM + A(I);
         end loop;**
```

In order to compute this information, the overlay function automatically converts the input sets into their corresponding regions of code. Most of these translations are automatically available (though heuristic in nature). In the case of the TPP, the user had to define that mapping at some time.

At this point, the user has identified a loop which computes the sum of the test scores. In order to find the initializations of this code, he views this region from the segmented parse perspective (where initializations are represented explicitly), and scans it for segments of the appropriate type. This is a filtering operation, in which the user applies restrictions to a previously identified set of objects.

```
> (Filter (Segs-Within CODE-REGION1)
          '(Seg-Type "initialization"))
     => SEGset2:[length 2]
```

The PRL converts CODE-REGION1 to a set of segmented parse frames (a heuristic process), and the function Segs-Within enumerates the subsegments it contains. The system identifies two initializations as a result. The user prints them by converting them to the textual view.

```
> (show! SEGset2)
   => for I in **1..N** loop
   => **SUM: REAL:= 0;**
```

The answers correspond to the initializations of the iteration variable "I", and the accumulation variable, "SUM". Note that the PRL retrieves the second initialization, even though it is lexically outside of the summation loop itself. It is identified from the segmented parse analysis, which associates a loop and its initializations no matter how far apart they might have been in the original code.

6. CURRENT STATUS AND FUTURE WORK

AI&DS is now developing a prototype version of the IPE (in a three year, 2-3 person effort), which is intended to demonstrate the efficacy of our knowledge based approach to the design of programming support tools. The prototype will embody a portion of all of the facilities that have been described. The IPE is currently targeted for the Ada language. It will initially run on a Symbolics 3600, a fast, personal LISP computer that features

a high-resolution bit-map display, but it is being designed to be portable to other systems (in particular, Unix).

We expect to augment the EPM's database to include more pragmatic information (e.g., the relation between requirements and program structures), and we intend to extend the PRL to the point where it will be able to automatically plan and carry out search requests of the kind demonstrated in this paper (based on a single user query). When these extensions are complete, the PRL will define a more formal reference language.

The task of building a prototype for the IPE involves a number of issues including the incremental modification of databases, and the recognition of user intentions in code. In order to solve these problems in the context of our applied research, we expect to rely heavily on methods for eliciting information from the user, and to focus on template-oriented techniques for manipulating programs.

Acknowledgements

We would like to thank Michael Brzustowicz and Eric Domeshek for their contributions to this project.

7. REFERENCES

1. Brotsky, D., Master's Thesis, MIT, forthcoming.

2. Dean, Jeffrey S., and Brian P. McCune, "Advanced Tools for Software Maintenance", AI&DS TR 3006-1, October 1982.

3. Nelson, T., "A New Home for the Mind," Datamation, March 1982.

4. Rich, Charles, "Inspection Methods in Programming", AI-TR-604, Artificial Intelligence Laboratory, MIT, 1981.

5. Shapiro, Daniel G., Brian P. McCune, and Gerald A. Wilson, "Design of an Intelligent Program Editor", AI&DS TR 3023-1, September 1982.

6. Teitelbaum, T., T. Reps, and S. Horwitz, "The Why and Wherefore of the Cornell Program Synthesizer", Proceedings, ACM SIGPLAN/SIGOA Conference on Text Manipulation, June 1981, pp. 8-16.

7. Waters, Richard C., "Automatic Analysis of the Logical Structure of Programs", AI-TR-492, Artificial Intelligence Laboratory, MIT, 1978.

8. Waters, R., "The Programmer's Apprentice: Knowledge Based Program Editing," IEEE Transactions on Software Engineering, SE-8, 1, January 1982, pp. 1-12.

APPENDIX B

This appendix contains a reprint of the paper "An Informal Study of Software Maintenance Problems," by Jeffrey S. Dean and Brian P. McCune.

[Software Maintenance Workshop, December 1983.]

AN INFORMAL STUDY OF SOFTWARE MAINTENANCE PROBLEMS

Jeffrey S. Dean
Brian P. McCune

Advanced Information & Decision Systems
201 San Antonio Circle
Mountain View, California 94040

ABSTRACT

A study of software maintenance problems was performed as the first step of a project aimed at suggesting advanced or novel techniques to increase reliability and reduce costs during the maintenance process. This paper summarizes some of the results of the study.

INTRODUCTION

In an effort aimed at finding long term solutions to the growing software maintenance problem, AI&DS conducted a two year software maintenance study for the Air Force [1]. The primary goal of this effort was to identify advanced tools and techniques (with particular emphasis on artificial intelligence techniques) capable of significantly impacting the software maintenance process within the next decade. The project was divided into three major phases: (1) studying the software maintenance process and identifying the major problems; (2) identifying tools and techniques; and (3) evaluating these tools and techniques. This paper summarizes our findings from the first phase of the the project.

WHAT IS MAINTENANCE?

For the purposes of this study, we used an "inclusive" definition of maintenance:

> Software maintenance is all those activities associated with a software system after the system has been initially defined, developed, deployed, and accepted as operational.

Maintenance is primarily a reactive activity: it is performed in response to requests (primarily requests for modification of software), rather than on the basis of some regular schedule.

This work was supported by Rome Air Development Center under contract F30602-80-C-0176.

OVERVIEW OF AIR FORCE SITES

To gain a better understanding of the problems encountered in large software maintenance environments, we studied the maintenance efforts at several Air Force C3I software organizations. The study consisted of one or more days of interviewing key personnel at each of the sites, followed by questionnaires being sent to these sites.

Characteristics of the three Air Force sites were collected during the interviews, and are summarized below.

Site 1:

application:	satellite tracking and control
software:	integrated system, coded in Jovial J4
size:	1 million lines
hardware:	network of small, medium, and large machines
developer:	outside contractors
maintainer:	ten different contractors
process:	batch processing, core patching

Site 2:

application:	communications
software:	numerous systems, generally coded in assembly language
size:	systems range in size from 25,000 to 560,000 lines of code
hardware:	variety of computers
developer:	outside contractors
maintainer:	in-house
process:	maintenance generally done in batch processing mode

Site 3:

application:	wide variety, from data processing to strategic planning
software:	numerous systems, coded in a variety of languages
size:	24 million lines of code
hardware:	wide range of computers
developer:	systems developed by outside contractors

maintainer:	mostly in-house, with some outside contractors
process:	outdated tools

THE SOFTWARE MAINTENANCE SURVEY

After the interviews, we sent out an informal survey to personnel at these sites. The purpose of the survey was to gather more information about the maintenance activities, to provide background and motivation for later phases of the project. No attempt was made to do as thorough or as statistically sophisticated an approach as other studies (such as [2]).

The survey was divided into three parts:

1. Reasons: "Why is software modified?"
2. Activities: "Where is time spent during maintenance?"
3. Problems: "Why is maintenance so difficult?"

Reasons for Software Modification

We divided modification requests into four categories:

1. Correcting: "There was something wrong with the software."
2. Adapting: "Something the system depended upon has changed."
3. Perfecting: "We wanted to fine-tune the system."
4. Modifying: "We didn't like the system the way it was."

These categories are similar to those in the Lientz and Swanson study [2], with the addition of one more category (modifying). Respondents were asked to estimate the percentage of requests that fell into each category. The averaged responses, in descending order, were as follows:

REQUEST	PERCENTAGE
modifying	46%
correcting	31%
perfecting	15%
adapting	8%

Requests in the modifying category alone account for almost half of the requests. Together with the perfecting category (the other category for "refinement" type requests), they account for over 60% of the requests (similar to [2]). Software maintenance has often been thought of as repairing software. However, these numbers indicate that software evolution (i.e., refining, as compared to repairing) is a significant part of the maintenance phase.

Software Maintenance Activities

We divided software maintenance into a number of activities, and asked respondents to rate the importance of each activity on a scale from 0 to 10 (with 10 signifying "extreme amounts of time spent on this task"). The averaged responses, in decreasing order, were as follows:

TASK	IMPORTANCE
testing	6.5
coding	6.3
training of new personnel and users	4.8
monitoring, problem detection, diagnosis	4.7
design	4.4
documentation	3.9
management	3.6
configuration control	3.4
analysis and specification of requirements	2.9

It is interesting to note that more time was spent on lower level tasks (such as testing and coding) than on higher level tasks (such as specification and design). Unfortunately, our survey did not probe sufficiently to determine the reasons behind this distribution of effort; we cannot tell if higher level tasks were neglected, or if lower level tasks were just inherently more time consuming. If higher level tasks are indeed being neglected, this would most likely have a negative impact on the overall maintainability of software.

Software Maintenance Problems

The last section of the survey identified four major software maintenance problems that were identified during interviews. Respondents were asked to rate the importance of each problem on a scale from 0 to 10 (with 10 signifying "extremely important problem"). These averaged responses, in descending order of importance, were as follows:

PROBLEM	IMPORTANCE
high turnover of personnel	8.7
understanding software/ lack of good documentation	7.5
determining relevant places to make changes	6.9
monitoring and diagnosing operations	6.3

The personnel turnover problem in the Air Force is the result of an average two year rotation cycle that causes a continuing, regular turnover.

THE COMPREHENSION PROBLEM

The top three maintenance problems all appear to revolve around a lack of understanding of the software and of the maintenance environment. We call this the comprehension problem. The relation of comprehension/understanding to these problems is clear:

- High turnover of personnel: Experienced personnel are replaced with new personnel who are unfamiliar with the applications software, and may be unfamiliar with the programming environment (tools, operating procedures, etc.) as well. The turnover rate is so high that there is little time allocated to update the documentation adequately.

- Difficulty in understanding software/lack of good documentation: Software to be maintained is hard to understand, particularly in the absence of current, high quality documentation.

- Determining all relevant places to make changes: Programmers have a hard time knowing where to make changes because they do not understand well enough how the code works.

ADVANCED TOOLS TO REDUCE THE COMPREHENSION PROBLEM

During the last two phases of this project, we identified nine tools/techniques for improving the maintenance process, and evaluated these ideas by another set of surveys [1]. The highest ranked tools address the problem of comprehension by explicitly collecting information about programs, documentation, and/or the programming process, and helping programmers apply that information on a regular basis.

CONCLUSIONS

The results of the survey shed light on three important issues in the maintenance process. First, most of the requests for maintenance are requests for refinement, rather than requests for repair. This reinforces the idea that maintenance is primarily a process of evolution. Second, most of the time spent is spent on low-level tasks, such as testing and coding. Finally, most of the difficulty in the maintenance process appears to arise from a lack of understanding of the application software, as well as the maintenance environment.

References

[1] Dean, J., and B. McCune, Advanced Tools for Software Maintenance. Rome Air Development Center, RADC-TR-82-313, December 1982.

[2] Lientz, B., and E. B. Swanson, Software Maintenance Management. Addison-Wesley, 1980.

[3] Shapiro, D., and B. McCune, "The Intelligent Program Editor: A Knowledge-Based System for Supporting Program and Documentation Maintenance," in Automating Intelligent Behavior: Applications and Frontiers. IEEE Computer Society, May 1983, pp. 226-232.

APPENDIX C.

This appendix contains a reprint of the paper "Trends for Advanced Software Tools," by Brian P. McCune and Jeffrey S. Dean.

[Invited paper, EASCON '83, September 1983.]

TRENDS FOR ADVANCED SOFTWARE TOOLS

Brian P. McCune and Jeffrey S. Dean

Advanced Information & Decision Systems
Mountain View, California

ABSTRACT

A recently completed study determined the major problems in the maintenance of Air Force command, control, communications, and intelligence software and proposed a number of advanced software tools to deal with these problems. Most of these advanced tools will rely on knowledge-based techniques from the field of artificial intelligence (AI). During the course of this research, a number of general trends were noted in the characteristics of these and other software tools, including both AI and non-AI tools. Among these trends are the use of knowledge of and reasoning about the domain of application, the performance of tool activities in small, incremental steps to provide better feedback to the programmer, the increasing intelligence of user interfaces to software tools, and the maintenance and use of a global knowledge base including a history of what has been done before and why. This paper discusses these and other trends for advanced software tools.

1. INTRODUCTION

The effort and expense of maintaining software have been recognized as major limitations on the capabilities of current software systems. The difficulties arise for several reasons. First, although hardware costs have decreased, software expenses have skyrocketed due to the higher cost of professional programmers. Second, as software projects have become more and more ambitious, the technical difficulty of making changes to the resulting programs has increased dramatically. As an illustration of this fact, the maintenance costs for large systems typically surpass the funds required for their initial development; the Department of Defense now spends more than three billion dollars per year on software maintenance. These problems are addressed in part by the creation of standardized structured languages such as Ada, but in our opinion they will only be solved by the results of new research into automated programming support systems. We expect that many such tools will rely on the application of artificial intelligence (AI) techniques.

To gain better insight into the specific problems of software maintenance, we performed a study which analyzed software maintenance problems in the Air Force [Dean & McCune-82]. The study concluded that the process of comprehending the form and function of existing software (i.e., what it does and how it does it) is the most crucial step in the maintenance process. A number of tools were defined, each of which could provide a limited operational capability in the short term (i.e., less than three years) and then gradually be enhanced in the medium term (i.e., three to seven years) and beyond.

This "comprehension problem" is revealed in many ways. To begin with, most programming installations have a high turnover rate of personnel and have trouble finding qualified replacements. As a result, maintenance personnel are often unfamiliar with the programs that are being maintained. At the same time, documentation is often unavailable or of poor quality. This increases the difficulty of comprehending a given program. It is not easy to understand a program by directly reading the code because of the quantity of detail involved and also because coding standards are poorly enforced and rarely agreed upon. Finally, the process of isolating bugs, designing solutions, and determining the ramifications of changes is difficult in the presence of an incomplete understanding of the program's organization. The relative difficulty of this task is affected by the tools available to the programmer.

The software maintenance study identified a collection of tools designed to alleviate these problems, all of which rely on a sophisticated understanding of the structure of programs. In effect, they operate by transferring some of the expertise currently in the minds of programmers into a machine-usable form that can be shared. Three of the most relevant tool ideas are summarized below. Advanced Information & Decision Systems is actively working on all three of these tools.

- The Programming Manager (PM) assists a programmer by systematically applying administrative

This research was supported in part by Rome Air Development Center under contract F30602-80-C-0176 and by the Air Force Office of Scientific Research under contract F49620-81-C-0067.

and technical policies. It enforces some procedures (e.g., testing of code before installation), suggests others (e.g., notifying a user group of a change), and automatically performs some actions on its own. In order to perform these functions, PM has a model of the underlying environment and each tool in the environment, including calling options and expected output. The Programming Manager is also intended to capture heuristic knowledge about code, for example, that bugs in module A often cause runtime errors in module B.

- The Intelligent Program Editor (IPE) is a knowledge-based tool for supporting the development and maintenance of software [Shapiro & McCune-83B]. It embodies a deep understanding of the structure of programs, of techniques for searching for relevant parts of programs based upon complex queries [Shapiro & McCune-83A], and of the manipulations that programmers typically apply to code. It can provide access to a variety of other tools that deal with code, e.g., the Documentation Assistant described below.

- The Documentation Assistant is a system that helps obtain, organize, access, and maintain many different forms of documentation, ranging from line-by-line comments to design principles and application-oriented requirements that underly the structure of the code as a whole. The Documentation Assistant is intenced to provide knowledge that other systems (such as the IPE) can employ.

2. TRENDS

In surveying existing production and research-prototype tools, as well as in our own research efforts, some particularly important trends and techniques have surfaced. These trends represent paradigms for the entire programming process, capable of forming the basis of a new generation of programming tools. Other than that, these trends are fairly dissimilar, varying in scope from the very broad to the fairly specific.

The remainder of this paper discusses these general trends that we see occurring now and into the future of software tool development. Definitions of specific tools that embody many of these trends are presented in [Dean & McCune-82]; three of these were mentioned above. We confine ourselves to a discussion of trends for tools, rather than underlying programming or related languages. We also assume that for at least the next decade programming environments and the programming process will evolve from the current state-of-the-art. Thus, we do not speculate on the potential of radical or revolutionary alternatives to programming, such as automatic programming [Elschlager & Phillips-82], in which a single monolithic tool hides all processing details from the user.

We discuss nine important trends in programming tools, programming environments, and their use. These trends are

1. Advanced capabilities
2. Domain knowledge and reasoning
3. Ability to be tailored
4. Life-cycle coverage
5. Tool integration
6. Advanced user interface
7. Integrated database
8. Incrementalism
9. Distribution

2.1 ADVANCED CAPABILITIES

An obvious trend in software tools is that toward more advanced capabilities. This arises in part from the continuing drop in hardware prices and increase in the demand and price of skilled programmers. It makes economic sense to automate more and more of the clerical programming functions as additional cpu cycles become cheaper than additional hours of human labor.

Probably more important in the long run than cost trade-offs of hardware versus people are the great technical advances that are on the horizon. Advances in a number of areas are going to have an important impact in the next decade. Among these technical areas are:

- Artificial intelligence. Artificial intelligence (AI) is the science and art of automating problem-solving processes that are informal, heuristic, and symbolic in nature. The simplest definition of AI is any activity that is performed by a non-human entity (typically a digital computer) and that is usually considered to require intelligence when performed by humans. At the core of AI are two notions: the complex manipulation of symbols (as opposed to numbers or text), and the use of heuristics ("rules of thumb") that can guide one quickly to a likely or satisficing solution. AI systems usually perform complex inferencing that involves combining the use of a number of heuristics in an appropriate fashion to solve a problem. Much of the research in applying AI to programming has concentrated on fully automating the process, except the specification stage. AI techniques such as heuristic reasoning, learning, natural-language understanding, and representation of domain knowledge may prove very useful when applied to today's programming environments. (AI is now being applied to numerous other defense problems, such as ocean surveillance [Drazovich, McCune, & Payne-82] and ship classification

[McCune & Drazovich-83].)

- Very high-level languages. A very high-level language (VHLL) is a programming language that provides capabilities significantly beyond the capabilities offered by traditional high-level languages. The level of a language refers to its similarity or closeness to machine language. Assembly language is a low-level language; it maps directly into machine language and requires the programmer to be familiar with the basic operations of the target machine. Languages like FORTRAN and PASCAL are considered high-level programming languages (HLLs); they provide the programmer with a computational model that is somewhat higher than machine level (e.g., by allowing the programmer to talk about variables and loops, instead of memory locations and jumps). Languages such as APL and LISP are considered even higher level, falling somewhere between HLLs and VHLLs; they allow the programmer to talk about arrays, lists, and the composition of operators. Experimental VHLLs exist that provide representation of sets and mathematical operations on them (e.g., SETL [Kennedy & Schwartz-75]), or objects and operations for specific application areas. VHLLs remain a research topic because the process of translating a VHLL program into an efficient program is difficult. VHLLs can still be effectively employed, by virtue of their ability to reduce manpower costs.

- Program transformation. Program transformation is the conversion of a program into another, computationally "similar" program, where the degree of similarity ranges from analogous to equivalent. Transformations may be done for a variety of reasons. If a program library contains a routine similar to what the programmer needs, it may be possible to automatically transform that routine into the desired one; if a program is written in a nonprocedural specification language, it may be necessary to transform the program into a more procedural form before it can be translated into some real programming language; if the program is written using inherently inefficient constructs, transformation can convert those constructs into more efficient ones. Taking the idea of transformation one step further, the entire programming process can be thought of as a series of program transformations or refinements, going from a high-level specification to the actual code.

- Formal verification. Formal verification is the demonstration that a piece of program is consistent with a given specification. This demonstration is carried out as a proof within the framework of a formal mathematical system that in most cases is based on first-order predicate logic. The specification formally describes desired properties of the program, it may give a complete specification of functional behavior (relationship between input and output values) or a specification of certain aspects, like absence of particular runtime errors, security of data flows, termination, or bounds on running time. Formal program verification is one form of program validation. It differs from others by requiring rigorous and formal specifications as well as the capability for reasoning about programs, and in turn provides a much higher degree of assurance that a program indeed performs as specified.

- Symbolic execution. Symbolic execution means evaluation of a program with symbolic values instead of actual data. Symbolic execution creates symbolic expressions that represent the values of outputs as a function of input variables, and (symbolic) predicates ("path conditions") that characterize the subset of values that cause the program to execute a particular program path. Symbolic evaluation thus shows the dependencies between the values of different variables and between data and control flows. Symbolic execution provides a versatile and powerful tool for debugging and analyzing programs. In comparison with ordinary testing, one symbolic execution of a program may correspond to a potentially large (even infinite) number of normal test runs. Symbolic execution may be considered a weak form of program verification; it shares some of the problems of verification systems.

- Graphics and other advanced input-output. Graphics and other forms of advanced input-output (I/O) are valuable in improving the user interface. For comprehension of complex information, graphical displays excel at helping users reach their potential. At worst, they can be used to mimic the linear textual output of hardcopy terminals; more appropriately, they can be used to display drawings and schematics as well as dynamic ("moving") pictures. Terminals with full-page, high-resolution displays are now available (e.g., Xerox STAR, Apple LISA, Symbolics 3600 LISP workstation). These allow the use of screen pages that may be as large as actual hardcopy pages; additional software provides the capability for stacking or overlapping these "windows".

- Software metrics. Measurements of performance are necessary to judge both programmers and software. Used appropriately, this data can be used to objectively improve the software process. For example, performance statistics for a programmer can be useful in determining appropriate training courses; statistics about the quality of a program can be used to help decide if the program should be modified or rewritten. Typical software metrics provide quantitative measures of program complexity. An example metric is the degree of interconnectivity of a set of modules as determined by an analysis of their data and control flow graphs. These measures can be used to predict estimated development or maintenance effort, to guide the development and maintenance process, or to predict the reliability (lack of errors) of a program.

- Computer-assisted instruction. The field of computer-assisted instruction (CAI) has been

attacking subjects such as logic and foreign languages, as well as more elementary topics, for some time. A small amount of work has been done on teaching particular programming languages and, to some extent, programming techniques. To speed up the learning cycle, the CAI system usually has access to the programming tools for the appropriate language, in order to compile and run programs and automatically grade performance by examining their output.

2.2 DOMAIN KNOWLEDGE AND REASONING

By domain we refer to an area of expertise, such as programming or a particular application area. Knowledge of and reasoning about a specific domain can be quite useful in a programming support environment. This is nicely illustrated with an example from an analogous situation. Suppose you had a technical manuscript that needed to be typed. If you gave the manuscript to a typist who spoke no English, you would expect, at best, a word-for-word typewritten copy of the manuscript. If you gave it to an English-speaking typist, you would hope that simple errors, such as misspellings and punctuation errors, would be fixed. If you gave it to an English teacher moonlighting as a typist, you wouldn't be surprised to find that some of your prose had been improved upon. And if you were lucky enough to find a typist familiar with the domain of discourse (of the manuscript), you shouldn't be surprised to find factual errors corrected.

The problem of getting the manuscript typed with the best possible result is similar to the problem of writing a program. You select some type of editor to use in entering the program text. A standard text editor would be comparable to the non-English-speaking typist: text is entered exactly as typed, with no enhancements. The English-speaking typist could be compared to a syntax-oriented editor, which can eliminate syntactic program errors and misspelled keywords (e.g., GANDALF's editor, MENTOR, Cornell Program Synthesizer). The other two typists have a fair degree of knowledge and understand how to apply it. The English teacher/typist knows about the language itself (rather than the content of what is being said). This situation is comparable to a programming language-specific editor, which applies knowledge about the domain of programming; the editor can help with general programming techniques, can catch certain types of semantic errors, can make style suggestions, and can improve the general flow of the program. The technical typist who understands the content of what is being said is analogous to an editor that utilizes knowledge about the application domain. It can help with domain-specific techniques, such as algorithm development, and can catch certain kinds of pragmatic errors that are dependent upon the specific application domain.

So, in a programming support environment, it is desirable to have two types of expertise, programming expertise and application expertise. An "ultimate" goal might be to endow the system with expertise equaling that of a human; the system would exhibit programming expertise comparable to that of a computer scientist and application expertise similar to that of a domain specialist. Note that in a programming support environment, the latter type of knowledge is more specialized, hence less widely applicable (a new knowledge base is needed for each application area).

The use of domain knowledge and reasoning in the programming environment will drastically change the whole concept of programming. It will allow the software tools to truly help the programmer, freeing the programmer to concentrate on higher-level issues.

2.3 ABILITY TO BE TAILORED

Future tools will have the ability to be highly tailored to suit the needs of the particular situation, including management hierarchy, application domain, other tools available in the programming environment, and idiosyncrasies of the tool users. Obviously, this level of variability goes well beyond the simple parameterization or runtime options found in many tools today. Modeling large bodies of facts and preferences requires knowledge representation techniques from AI. Modifying these bodies requires an ability to elicit new or modified knowledge from users or to learn by observation. These areas are among the most difficult in AI research, but the potential is tremendous.

2.4 LIFE-CYCLE COVERAGE

Future environments will have capabilities that support more of the software life-cycle. Most important, maintenance of software after initial release is recognized as typically requiring two-thirds of the overall lifetime costs of a software system. Therefore tools must be designed with maintenance, as well as development, in mind. Some tools may be developed solely for use in the maintenance phase.

The use of tools to date has been concentrated in the coding and testing phases of software development. There is an obvious reason for this: source and executable code and data are often the only forms of information stored in the computer and therefore available to tools. This situation is slowly changing, as other forms of information are formalized and automated, ranging from requirements and design specifications, to formal documentation and test data specifications, to management schedules and methodology descriptions, to measurements gained by applying software metrics. For each new type of information, tools are needed to assist in its creation, analysis, and transformation into other types of information.

Because so much more information than just code will be dealt with by tools, many future tools will be independent of a specific programming language.

2.5 TOOL INTEGRATION

There is a saying that "the whole is more than the sum of its parts". This notion of synergy is important in the design of software tools. When several tools work together, they may provide something that neither one could alone provide. The term integration is used here to refer to the degree of synergy and close coupling between tools. Tools in a well-integrated system exhibit a large degree of synergy (as a result of working well together). Since synergy results from interdependencies, integrated tools are likely to share information, share common procedures, or provide complementary functions. Systems such as INTERLISP [Teitelman & Masinter-81] and UNIX [Kernighan & Mashey-81] owe a large amount of their success to their integration.

Well-integrated systems provide several important advantages:

- Human comprehension is aided by the uniformity provided by a well-integrated system. A consistent underlying philosophy aids users in making inferences about how the system works.

- An integrated tool set allows one to put tools together quickly in order to perform tasks that may not have even been envisioned by the system designers. This benefit is well known to UNIX users.

- Integrated tools work together, allowing more efficient and effective performance. Efficiency is gained when one tool can make use of another's work, eliminating redundant computations; for example, symbol tables created by a compiler can be used by debuggers, linkers, cross-reference listers, etc. Effectiveness is increased when tools can make use of each others' information; for example, a compiler may be able to apply optimizing code generation strategies by getting information from a program verifier that the compiler cannot deduce by purely syntactic means.

Related to integration is the idea of completeness. Completeness means that the user should be able to do everything that might be needed. The beauty of an integrated system is marred when a user has to expend a large amount of energy to do something that is conceptually simple but that isn't allowed by the system. For example, the INTERLISP system allows certain common monitor-level commands to be performed without leaving the system. To perform other commands, there is a simple interface that creates a new process running the operating system's command processor, allowing the user to execute arbitrary commands and then return to INTERLISP without any loss of continuity.

Designing a consistent yet usable system requires a great deal of ingenuity and insight on the part of the designers. But the effort does pay off. Consider the popularity of UNIX. The Stoneman requirements for Ada Program Support Environments (APSEs) also specify an integrated tool set; the success of this requirement will be seen as APSEs are implemented and used.

2.6 ADVANCED USER INTERFACE

Future tools will have very advanced interfaces to programmers and other users. The most visible part of the interface is the collection of I/O techniques available. Input techniques may range from selection of commands from menus, pointing using a mouse or other cursor positioning device, natural-language input, and speech input. Output techniques include high-resolution graphics that are capable of displaying publication-quality program listings, the use of color to aid in focusing attention, and speech output.

Despite the current interest in advanced I/O techniques, the use of these techniques alone cannot solve software development and maintenance problems. The primary difficulty is in deciding what information to communicate to the programmer, rather than how to communicate it. A tool that uses some combination of advanced I/O techniques can be changed to work with simpler methods, usually without a significant loss in information or functionality.

Many tools converse with the programmer interactively. To be used effectively, it is necessary for the user to understand what the tool is saying and how to respond to it. On-line help facilities can teach generic command structures, but tools must also be able to explain the details of the current situation. Knowledge-based AI systems are generally capable of explaining their current state and what chain of reasoning got them there.

Going even further, advanced user interfaces should provide some of the intelligence and assistance that a human programming assistant might provide. An intelligent user interface not only makes life easier for the programmer; it helps increase programmer productivity and software reliability. Here are some of the kinds of features that an intelligent user interface might provide:

- Programmability. The user interface in a programming support environment should provide the programmer with tools for automating his own tasks, either by the programmer explicitly programming the tasks or by the system learning.

- Error prevention. By making "bad" things hard to do, it is less likely that they will be done inadvertently. Warnings about dangerous actions, before they are performed, further reduce the chance of error.

- Error detection and correction. It is not too difficult to catch many types of errors automatically. Every attempt should be made to catch errors as early as possible; the later an error is detected, the more expensive it is to fix. Error diagnostics should be meaningful to the user, not only to the person who wrote them. Some errors, especially silly, careless

ones (e.g., spelling errors), can be corrected without too much difficulty.

- Recoverability. If an error is made, the user should be able to recover as easily as possible. The system should have safeguards to provide the user with certain paths of recourse, e.g., by allowing actions (such as deleting a file) to be undone. "Forgiveness" is important. Making a blunder is bad enough; one should not have to spend hours or days to right it.

- Active help. If the user repeatedly does things incorrectly, there may be no need to wait until help is requested. In many cases, the user may not be aware that help exists, or may not know how to ask for it. Help should be offered automatically.

- Non-interactive operation. The system should be able to function without human intervention if necessary. If a programmer leaves the terminal while performing a task, there is oftentimes no need to bring things to a halt when only non-crucial human input is needed.

In order to accomplish these goals, an advanced user interface must have models of both the user and the process by which tools are used. It is necessary to understand the programmer's actions (what he is doing) and intentions (what he will be or wants to be doing). For example, an editor incorporating programming domain knowledge needs to know what parts of a program the programmer will be writing or changing, as well as the (expected) effect this will have on other parts of the system. An editor incorporating application domain knowledge needs to know what techniques the programmer will be utilizing, as well as the type of output and results expected. As soon as an environment has a model of what tools are available and how to access them, it is feasible to construct comprehensive or meta-tools, tools that reason about and invoke other tools on behalf of the user.

To help the programmer make decisions about what to do, tools need to understand the programming process itself in order to determine what the programmer is doing right or wrong. When a specific methodology is chosen for an environment, software tools should be provided to aid each step of the methodology.

2.7 INTEGRATED DATABASE

Information in most programming environments is stored as a set of individual files of various types. This is essentially just a classical file system as provided by most operating systems. The next generation of environments will probably use something closer to a relational model of data, so that uniform random access is possible to all objects and so that complex relational objects such as structured documentation can be easily stored and accessed. As more AI-based tools are incorporated into an environment, the database will evolve into a full-fledged knowledge base that incorporates all of the previous information plus complex semantic models and sets of heuristics.

The history list or audit trail is one database component that is recognized as important by most state-of-the-art environments (e.g., INTERLISP, APSEs). The notion of computational history refers to the information available during the course of some computation. For example, when using a text editor, the history includes the editing commands as well as the inserted and/or deleted text; when using a compiler, the history includes the original source code, the parse trees, parse tree transformations, and generated code. Some of this information has no long-term value beyond immediate consumption by a program; but much of the information is quite valuable, either because it is expensive to recompute (e.g., parse trees for a large module) or because it cannot be recomputed (e.g., a record of all operations performed by the user).

There are numerous reasons why history is a necessary ingredient in advanced programming environments. First of all, sophisticated programming environments must allow programmers to make changes incrementally, so that the cost of making small changes is small. To accomplish this, intermediate results of various system tools and utilities (e.g., compilers, linkers) must be kept around. Another need is accountability: records of all important activities should be maintained so it can always be determined what has been done and who has done it. Important activities include things like changes to code, document updates, system builds, etc. From the perspective of the user interface, preservation of a history is also desirable. Some programming systems, such as INTERLISP, allow the user to see a record of what has been done and allow transactions to be "replayed". Finally, history is necessary for the application of programming domain knowledge and reasoning: to understand what the programmer is doing, it is necessary to understand the context in which the programmer has been working.

2.8 INCREMENTALISM

Support of incremental change is vital for the maintenance of all but the smallest systems. It is unacceptable and unnecessary to require a whole system to be rebuilt each time a small change is made: unacceptable because the cost is too high, unnecessary because changes usually leave many parts of the system unaffected.

The move toward building systems that handle incremental change has been slow, primarily since it is (in general) more difficult to build tools that are incremental. There are several problems. First of all, new algorithms may have to be devised or old "batch" algorithms modified, in order to handle incremental requests. Another problem is lack of information: most tools throw out information as soon as they are done with it, rather than leaving it around for future reference. An example of this is symbol table information, which the

compiler builds up for each module and then usually discards. This means that the symbol table must be rebuilt for each recompilation, even if code changes had no effect on it.

Incrementalism is a technique vital for the development and maintenance of large systems, yet few existing programming tools make use of it. Aside from some research on incremental techniques for syntactic parsing, most attempts at incorporating incrementalism have been somewhat ad hoc (and less than generally applicable). The idea of incrementalism falls out naturally when some of the other techniques discussed earlier in this section (e.g., history) are incorporated into the programming environment.

2.9 DISTRIBUTION

As more of the non-coding functions of the software life-cycle are automated, it becomes clearer that the model of a single programmer interacting with a unique copy of the environment and database is not adequate. Large programming projects have numerous individuals operating asynchronously. These personnel may have different functions (e.g., supervisor, designer, coder, tester, documenter), different physical locations, different "home" computers, etc. Thus, programming environments are fast entering the era of distributed systems and processing, with all of the standard problems of planning and coordination, synchronization of computer objects and events, maintenance of multiple copies of objects, etc.

A number of architectures are possible for distributed programming environments. The simplest has each programmer accessing the primary development computer (probably a mainframe) via a front-end computer (probably an advanced personal computer). A few of the more interactive tools (e.g., editor, language interpreter) would run on the front-end, while most would remain on the mainframe (e.g., optimizing compiler, database handler). A more distributed architecture would have copies of each tool at each node, with no node being central. Finally, individual tools may also be distributed across processors someday.

3. CONCLUSION

We have presented nine trends that cut across many possible programming tools and support environments. Each trend will eventually require the use of AI to achieve its potential. In this role, AI is not a radical, high-risk approach to the software problem, but a technology that will make possible major enhancements to every aspect of the programming paradigm of today.

4. ACKNOWLEDGMENT

This paper has benefited from discussions with Gerald A. Wilson and Daniel G. Shapiro.

5. REFERENCES

- [Dean & McCune-82] Jeffrey S. Dean and Brian P. McCune, Advanced Tools for Software Maintenance, Technical Report 3006-1, Advanced Information & Decision Systems, Mountain View, California, October 1982.

- [Drazovich, McCune, & Payne-82] Robert J. Drazovich, Brian P. McCune, and J. Roland Payne, "Artificial Intelligence: An Emerging Military Technology," invited paper, Conference Record, EASCON '82: Fifteenth Annual Electronics and Aerospace Systems Conference, Institute of Electrical and Electronics Engineers, Inc., Washington, D.C., September 1982, pages 341-348.

- [Elschlager & Phillips-82] Robert A. Elschlager and Jorge V. Phillips, editors, "Automatic Programming", in Avron Barr and Edward A. Feigenbaum, editors, The Handbook of Artificial Intelligence, Volume 2, Chapter 10, William Kaufmann, Inc., Los Altos, California, 1982, pages 295-379.

- [Kennedy & Schwartz-75] K. Kennedy and J. Schwartz, "An Introduction to the Set Theoretical Language SETL", Computers and Mathematics with Applications, Volume 1, Number 1, 1975, pages 97-119.

- [Kernighan & Mashey-81] Brian W. Kernighan and John R. Mashey, "The UNIX Programming Environment", Computer, Volume 14, Number 4, April 1981, pages 12-24.

- [McCune & Drazovich-83] Brian P. McCune and Robert J. Drazovich, "Radar with Sight and Knowledge", invited paper, Defense Electronics, Volume 15, Number 8, August 1983.

- [Shapiro & McCune-83A] Daniel G. Shapiro and Brian P. McCune, Searching a Knowledge Base of Programs and Documentation, Technical Memorandum 1014-2, Advanced Information & Decision Systems, Mountain View, California, January 1983.

- [Shapiro & McCune-83B] Daniel G. Shapiro and Brian P. McCune, "The Intelligent Program Editor: A Knowledge-Based System for Supporting Program and Documentation Maintenance", Automating Intelligent Behavior: Applications and Frontiers, Proceedings, Trends and Applications 1983, IEEE Computer Society, Los Angeles, California, May 1983, pages 226-232.

- [Teitelman & Masinter-81] Warren Teitelman and

Larry Masinter, "The INTERLISP Programming Environment", Computer, Volume 14, Number 4, April 1981, pages 25-33.

END

FILMED

2-85

DTIC